Metody głębokiego uczenia, projekt nr 1

Implementacja algorytmu wstecznej propagacji błędu wperceptronie wielowarstwowym (MLP)

Tomasz Klonecki Tomasz Radzikowski

19 marca 2019

Wstęp

W projekcie stworzyliśmy program do obsługi sieci neuronowej, którą sami zaimplementowaliśmy. Sieć uczy się metodą wstecznej propagacji błędu. Podczas obsługi programu możemy podać kilka zmiennych które definiują charakterystykę sieci: - Liczba warstw i neuronów w każdej warstwie - Funkcje aktywacji w każdej warstwie - Liczbę iteracji (epok) - Wartość współczynnika nauki - Wartość współczynnika bezwładności

Funkcja błędu wykorzystywana w naszym algorytmie to MSE (Mean Squared Error).

Funkcja aktywacji warstw ukrytych jaką wykorzystaliśmy w testach to funkcja sigmoidalna:

$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$

- przyjmuje wartości od 0 do 1 co jest bardzo użyteczne, gdyż zawsze uzyskujemy znormalizowane wyniki.

Oczywiście w miarę potrzeby można do pliku *activations.py* dopisać nową funkcję.

Poniżej zaprezentujemy uzyskane wyniki na podstawie przykładowych danych i przeanalizujemy dlaczego sieć dobrze działa dla niektórych parametrów a dla niektórych nie działa w ogóle.

Instrukcja obsługi programu:

1. Tworzymy obiekt klasy **Network** i podajemy parametry sieci

`brain = Network(learning_rate = 0.001, momentum_rate = 0.7, iterations = 100)` 2. Do obiektu klasy **Network** możemy dodać dowolną ilość warstw pochodzących z klasy **Layer**: - **inputs:** liczba sygnałów wejściowych do warstwy - **outputs:** liczba sygnałów wyjściowych z warstwy - **activation_func:** funkcja aktywacji warstwy (na ten moment do wyboru mamy: sigmoid, softmax, linear)

```
brain.add(Layer(inputs = 1, outputs = 2, activation_func = 'sigmoid'))
```

3. Uczenie sieci odbywa się za pomocą jednej z dwóch wybranych metod uruchomionych na klasie Network
   - Batch training (full data set training)
     `brain.train_and_evaluate(x[0],x[1],x[2],x[3])`
     - **x** to lista zwracana przez poniższe funkcje do wczytywania danych
   - Mini batch training (stochastic gradient descent)
     `brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)`
4. Pokazanie wyników dla nauczonej sieci

Należy wykorzystać opisane niżej funkcje do rysowania wykresów

Oprócz tego napisaliśmy też kilka funkcji pomocniczych do wczytywania danych i do rysowania wykresów:

Funkcje do wczytywania danych

`data_read_classification(name = 'three_gauss', size = 100)`

`data_read_regression(name = 'linear', size = 100)` - **name:** rodzaj zbioru danych - **size:** wielkość zbioru danych

Funkcje do rysowania wykresów

`plot_classification(Network = brain, data = x)` - **Network:** obiekt klasy network - **data:** dane zwrócone przez powyższe funkcje do wczytywania danych

`plot_regression(Network = brain, data = x)` - **Network:** obiekt klasy network - **data:** dane zwrócone przez powyższe funkcje do wczytywania danych

`plot_classification_mesh(Network = brain, data = x)`

Funkcja do rysowania zbioru testowego jako siatki. Możemy dzięki temu lepiej zobaczyć prawidłowe dopasowanie. Tam gdzie mniej intensywny kolor nie zgadza się z bardziej intensywnym znaczy że dopasowanie nie zostało dobrze przeprowadzone. - **Network:** obiekt klasy network - **data:** dane zwrócone przez powyższe funkcje do wczytywania danych

`test_lr(network, x, lrate , momentum = 0.9, iterations = 100)`

Funkcja do testowania paramteru **Learning rate**. - **Network:** obiekt klasy network - **data:** dane zwrócone przez powyższe funkcje do wczytywania danych - **lrate:** np.array zawierający różne learning rate do przetestowania np. $lrate = np.random.uniform(0.00001, 0.1, 10)$ - **momentum** i **iterations** są domyślnie ustawione

`plot_errors(errors)`

Funkcja do rysowania wykresu funkcji błędu na zbiorze treningowym i testowym.

- **errors** - dane zwrócone przez funkcję **train__and__evaluate** lub **train__mini__batch__and__evaluate**

`weights_norms_plot(errors)`

Funkcja pozwala na rysowanie miary Frobeniusa macierzy wag, w jaki sposób zmieniały się ona podczas wszystkich epok. Wykorzystujemy ją przede wszystkim do tego, żeby kontrolować czy wagi nie rosną za szybko do nieskończoności. - **errors** - dane zwrócone przez funkcję **train__and__evaluate** lub **train__mini__batch__and__evaluate**

1. Wczytanie bibliotek, plików klas i funkcji

```
from network import Layer
from network import Network
from program_functions import data_read_classification
from program_functions import data_read_regression
from program_functions import plot_classification
from program_functions import plot_regression
from program_functions import plot_errors
from program_functions import plot_classification_mesh
from program_functions import test_lr
from program_functions import weights_norms_plot

import numpy as np
```

Problem regresji

W problemie regresji zastosowaliśmy jedną praktyczną zmianę, czyli wykorzystanie funkcji liniowej $f(x) = x$ jako funkcji aktywacji w ostatniej warstwie sieci. Powodem jest potrzeba uzyskania na wyjściu sieci wartości z $\mathbb{R}^3$.

Funkcja liniowa

Pierwszym problemem, z którym mierzy się nasza sieć jest rozwiązanie zagadnienia regresji liniowej. Stopień skomplikowania tego zadania zależy w dużej mierze od danych wejściowych, bowiem dla prostych funkcji nawet bardzo prymitywna sieć o małych rozmiarach i przy niskiej liczbie iteracji spełnia swoją funkcję w sposób dostatecznie dobry. Unaocznieniem powyższego stwierdzenia jest wykres nr 1, gdzie dla dwuwarstwowej sieci $(1, 2)$ i 10 iteracji uzyskujemy niski błąd.

```
np.random.seed(0)
x = data_read_regression('linear',100)

# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3
brain = Network(learning_rate = 0.0001, momentum_rate = 0.8, iterations = 10)
brain.add(Layer(1,2,'linear'))
brain.add(Layer(2,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 10/10
```

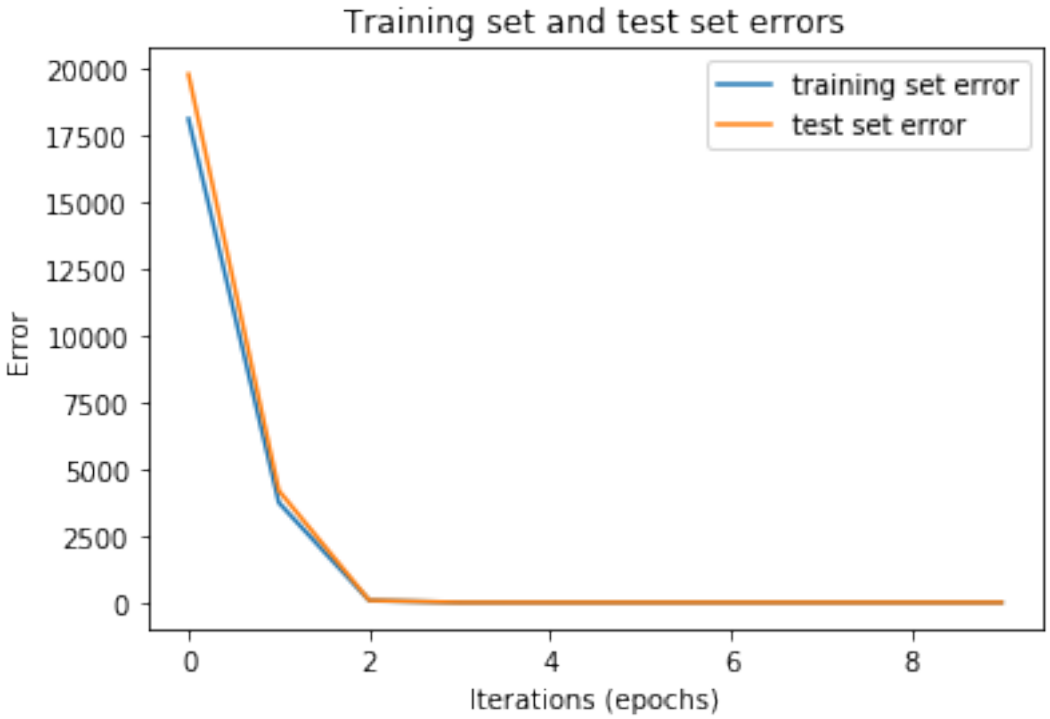

Figure 1: png

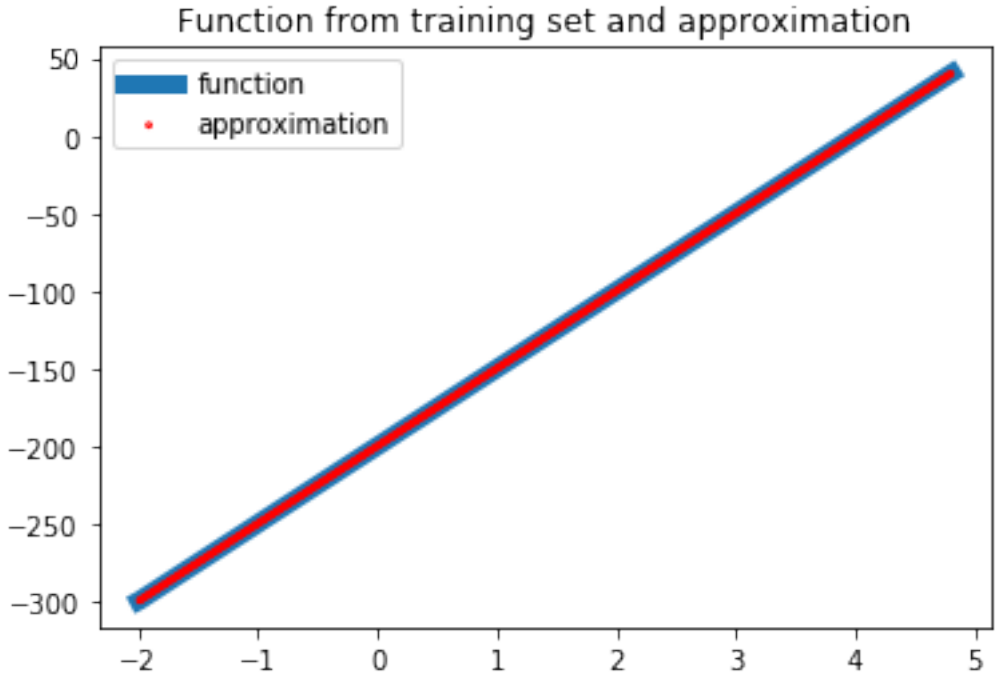

Figure 2: png

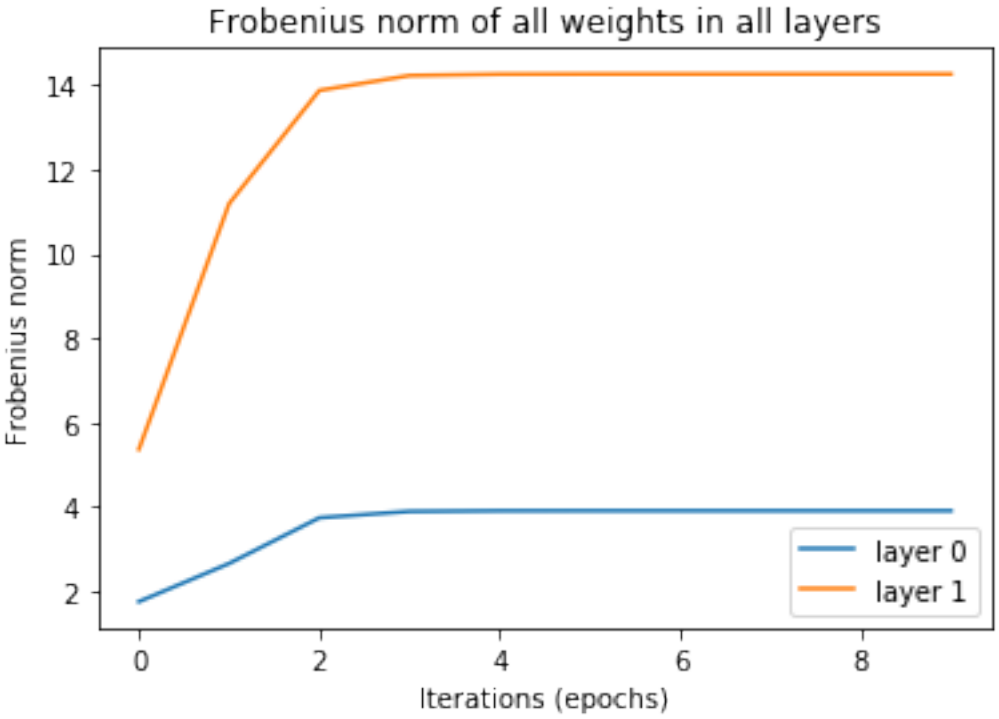

Figure 3: png

Funkcja kwadratowa

Jednak już dla dla funkcji kwadratowej błąd rośnie kilka tysięcy razy, a dopasowanie funkcji aproksymacyjnej (co jest absolutnie naturalne, biorąc pod uwagę, że sieć wykorzystuje funkcję liniową) jest bardzo niedokładne. Prosta zmiana funkcji aktywacji na pierwszej warstwie na funkcję sigmoidalną sprawia, że dopasowanie, choć nadal bardzo złe daje nadzieję na lepsze dopasowanie przy modyfikacji parametrów sieci.

```
np.random.seed(0)
x = data_read_regression('square',100)

# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3

brain = Network(learning_rate = 0.001, momentum_rate = 0.8,iterations = 200)
brain.add(Layer(1,2,'sigmoid'))
brain.add(Layer(2,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 200/200
```

Figure 4: png

Samo dodanie neuronów w drugiej warstwie sprawia, że funkcja błędu jest o wiele gładsza i nie oscyluje między wartościami, błąd jest mniejszy dziesięciokrotnie.

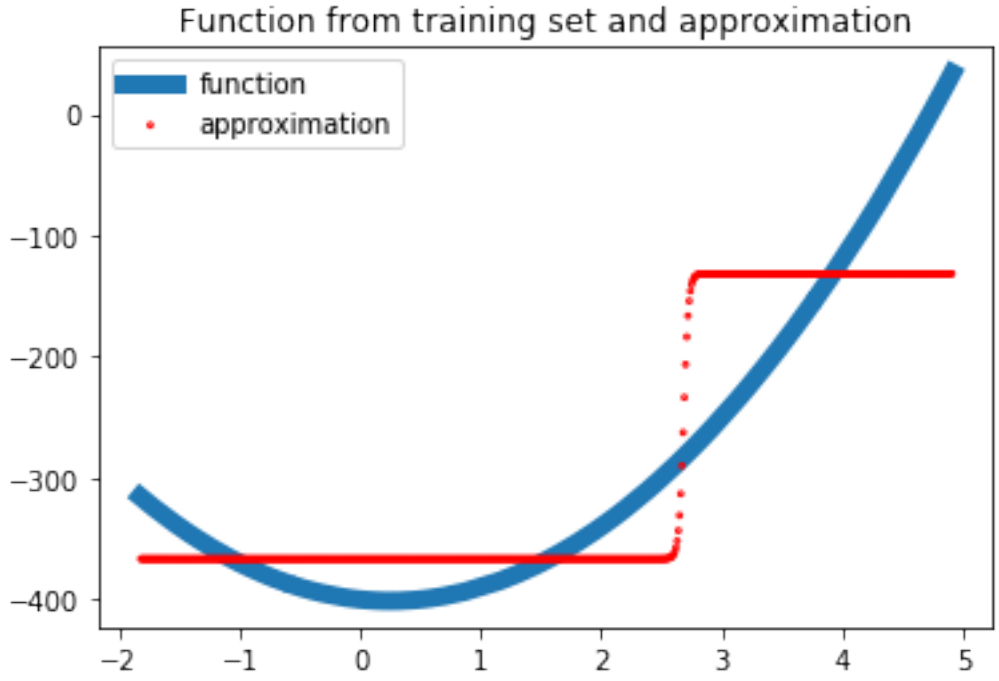

Figure 5: png

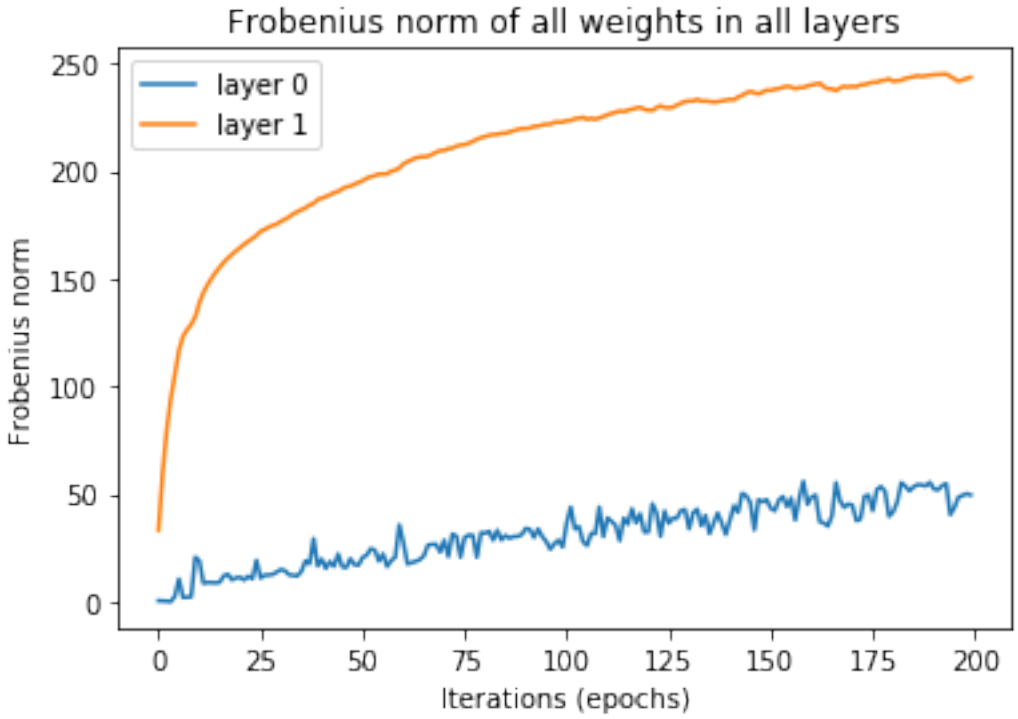

Figure 6: png

```
np.random.seed(0)
brain = Network(learning_rate = 0.0001, momentum_rate = 0.8, iterations = 30)
brain.add(Layer(1,20,'sigmoid'))
brain.add(Layer(20,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 30/30
```

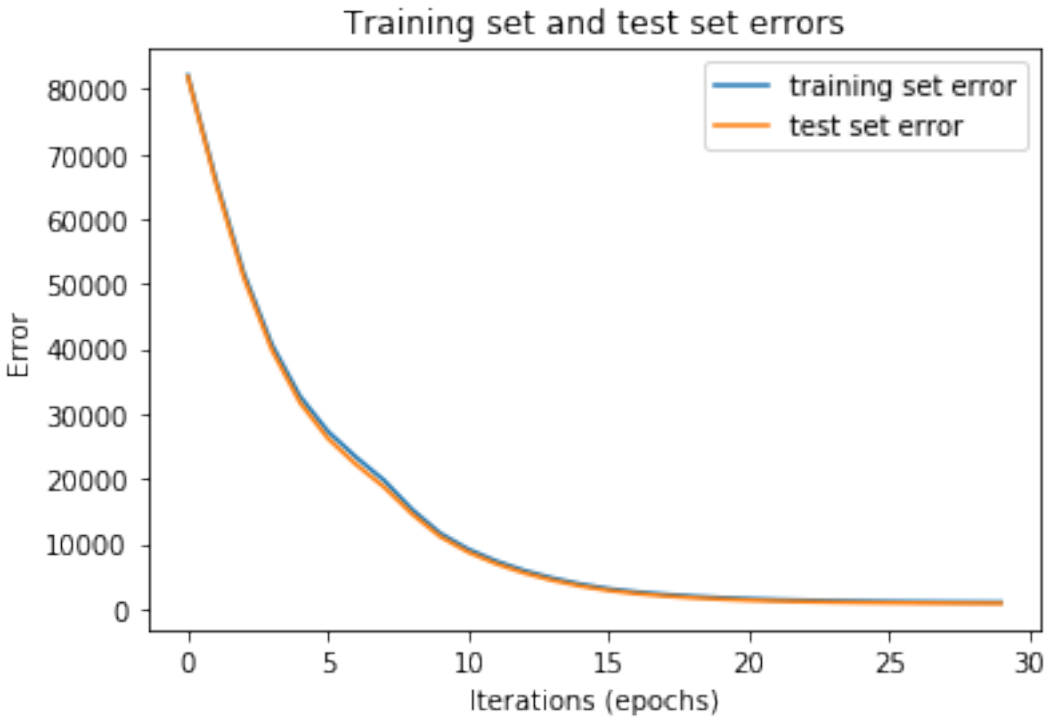

Figure 7: png

Dostatecznie dobre dopasowanie uzyskano, zwiększając liczbę iteracji dla sieci o trzech warstwach. Przebieg funkcji jest odwzorowany oprócz małych fragmentów na początku i na końcu wykresu.

```
np.random.seed(1)

brain = Network(learning_rate = 0.0001, momentum_rate = 0.2, iterations = 200)
brain.add(Layer(1,150,'sigmoid'))
brain.add(Layer(150,80,'sigmoid'))
brain.add(Layer(80,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 200/200
```

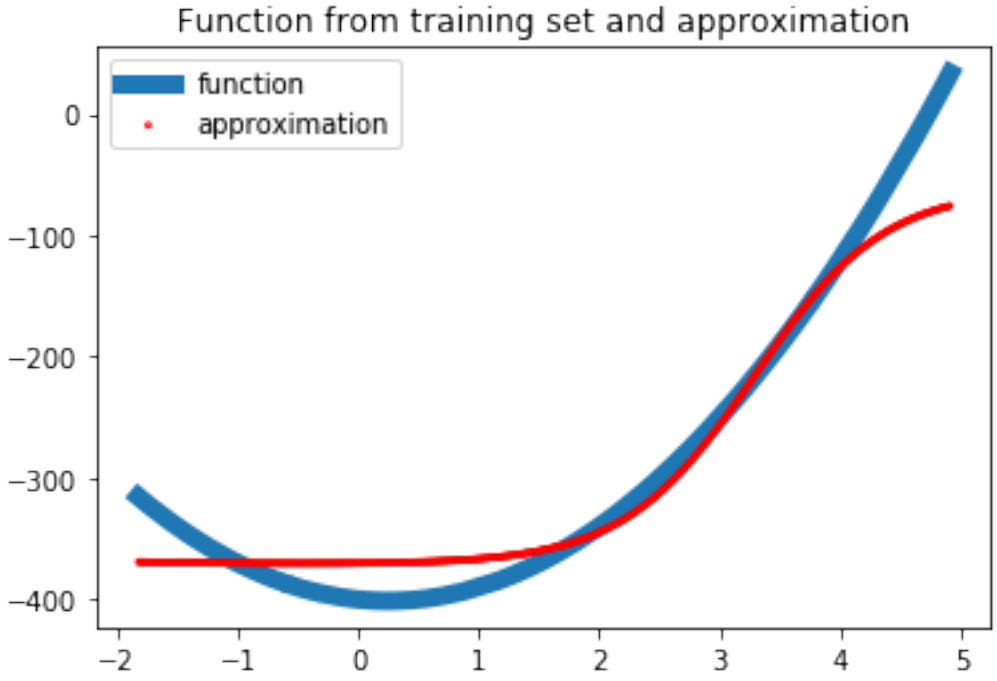

Figure 8: png

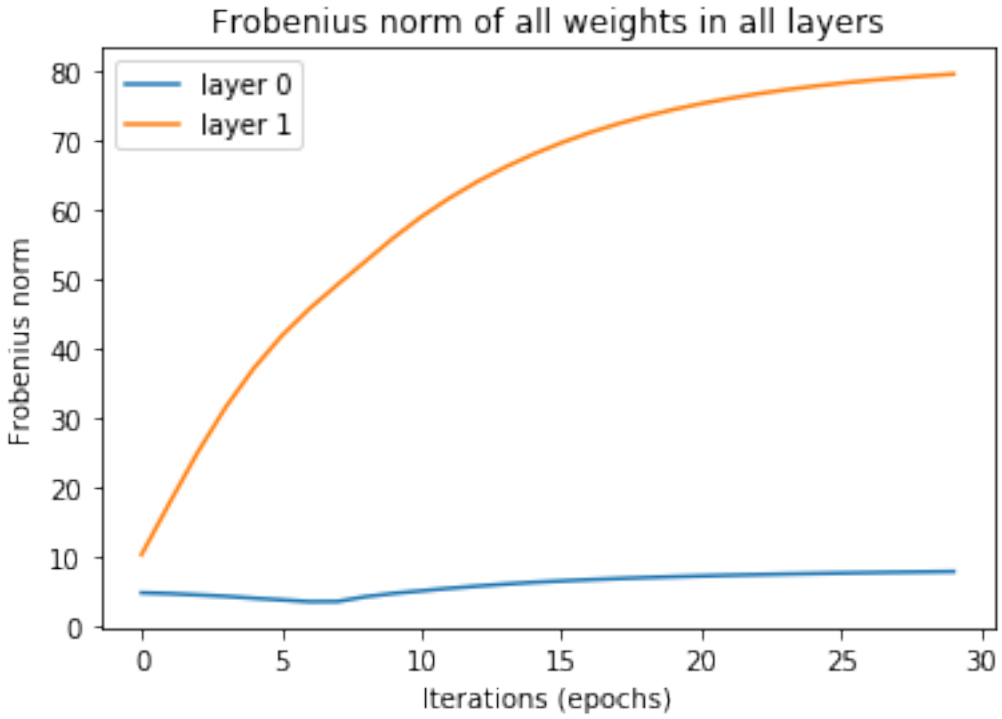

Figure 9: png

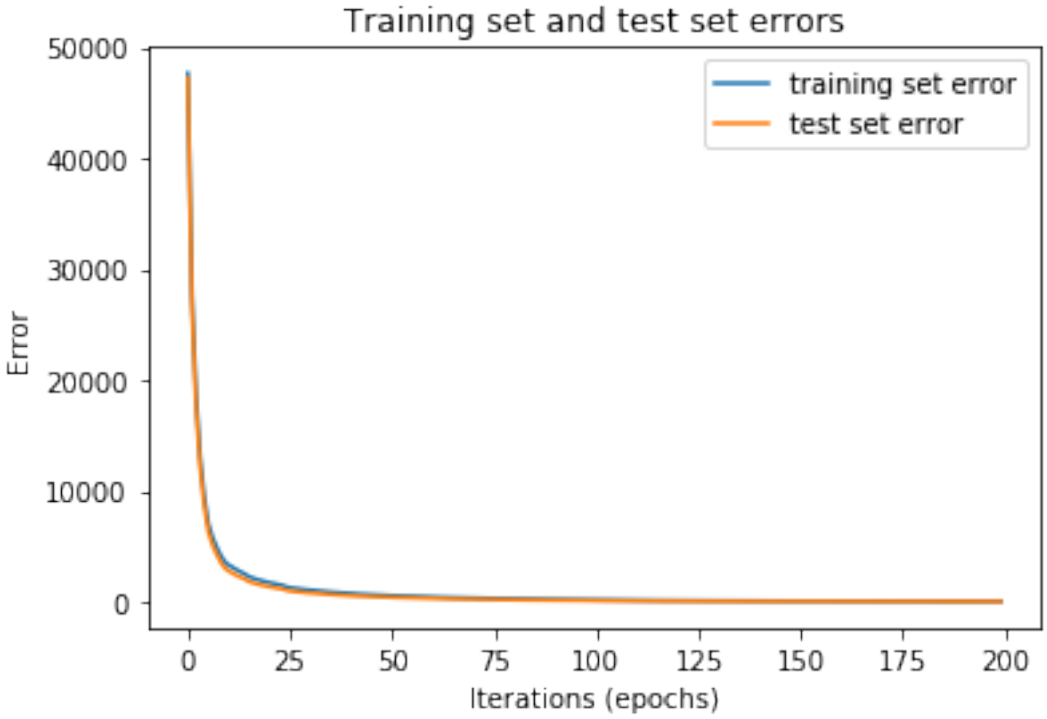

Figure 10: png

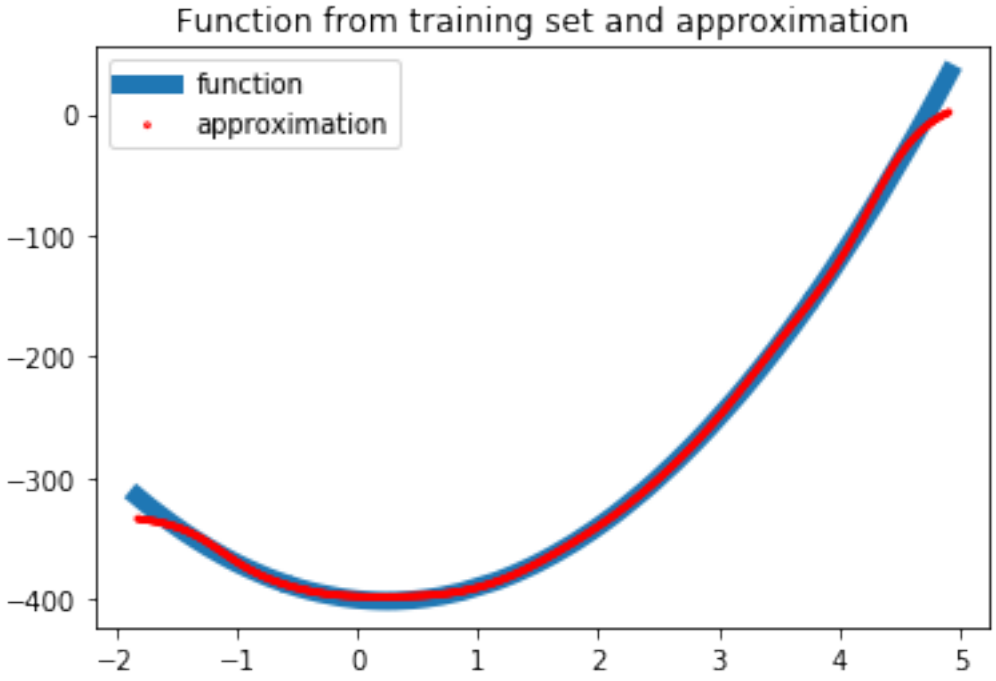

Figure 11: png

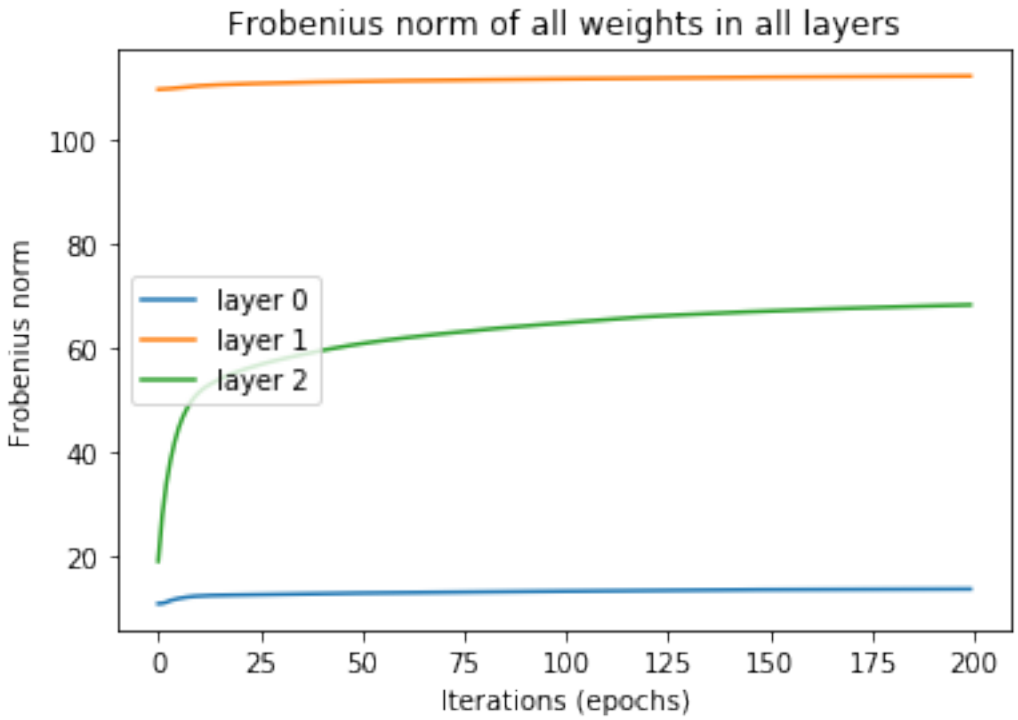

Figure 12: png

Funkcja wielomianowa stopnia 3-ciego

Dla tego typu funkcji dobre rezultaty dało zwiększenie liczby iteracji i dodanie jeszcze jednej warstwy sieci. Ewentualną poprawę dopasowania można uzyskać w prosty sposób, poprzez dodanie kolejnych iteracji.

```
np.random.seed(0)
x = data_read_regression('cube',100)

# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3

brain = Network(learning_rate = 0.0001, momentum_rate = 0.2, iterations = 500)
brain.add(Layer(1,150,'sigmoid'))
brain.add(Layer(150,200,'sigmoid'))
brain.add(Layer(200,80,'sigmoid'))
brain.add(Layer(80,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 500/500
```

Kolejnym problemem, który rzutuje na jakość dopasowania jest wielkość zbioru treningowego. Rozpatrywany uprzednio przypadek o n=100 wymagał dużej sieci. Dla n=1000 znacznie mniejsza

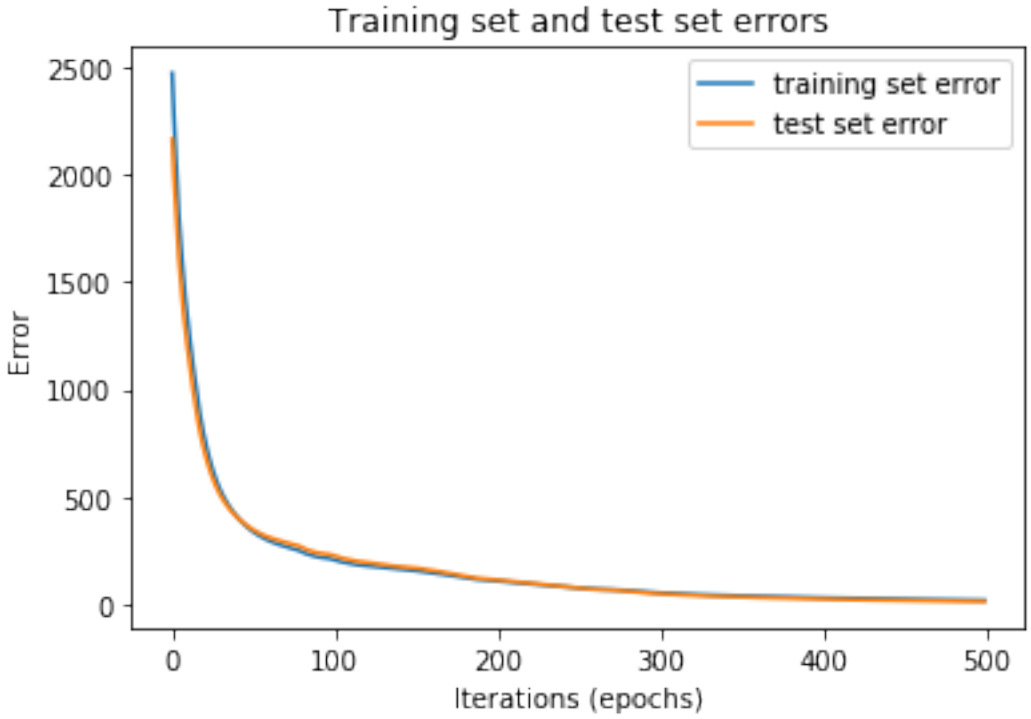

Figure 13: png

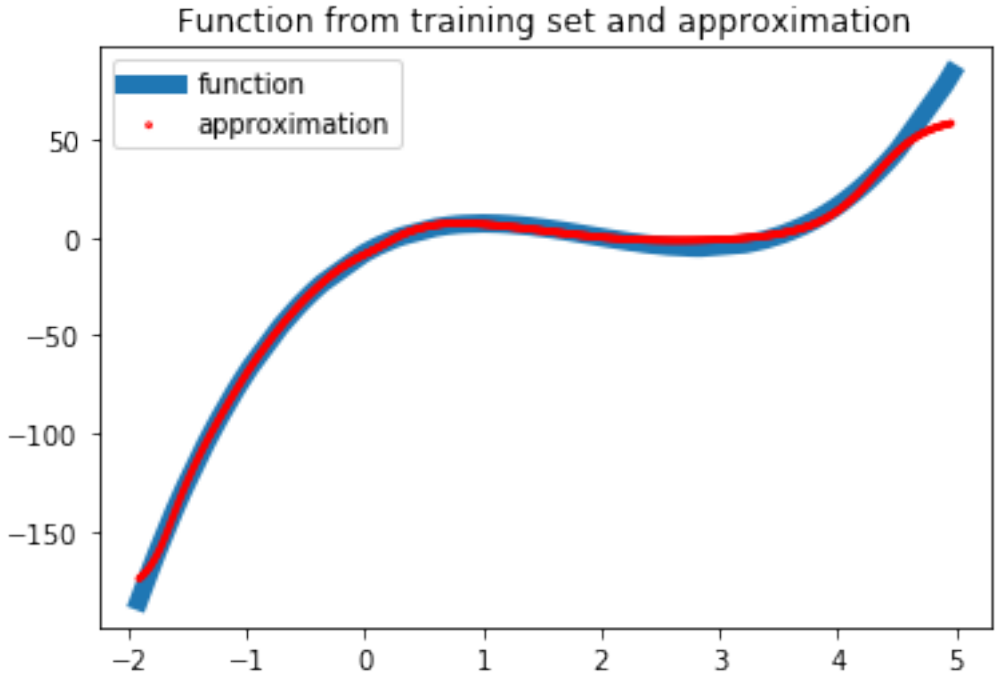

Figure 14: png

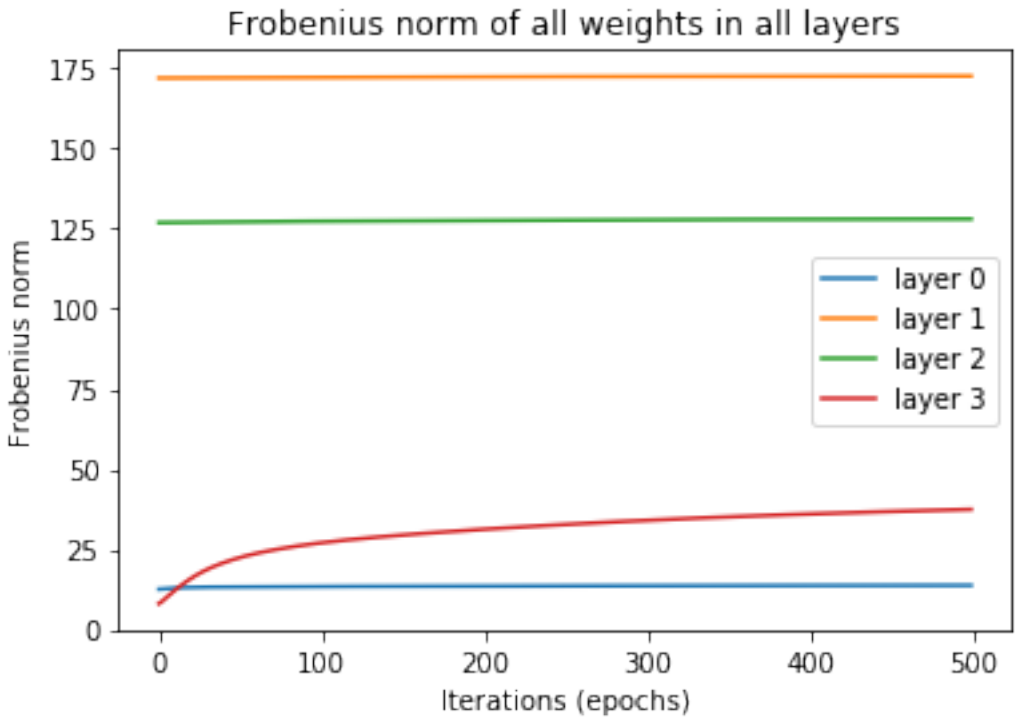

Figure 15: png

sieć radzi sobie z dopasowaniem.

```
np.random.seed(0)
x = data_read_regression('cube',10000)

# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3

brain = Network(learning_rate = 0.001, momentum_rate = 0.8, iterations = 100)
brain.add(Layer(1,30,'sigmoid'))
brain.add(Layer(30,15,'sigmoid'))
brain.add(Layer(15,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
#errors = brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Funkcja aktywacji

Dla funkcji aktywacji zachodzi ciekawe zjawisko, gdyż taka sama klasa funkcji jest reprezentowana w pierwszej warstwie sieci (funkcja sigmoidalna). W związku z tym, sieć z jednym neuronem w warstwie wejściowej, przy odpowiedniej liczbie iteracji pozwala na uzyskanie dopasowania.

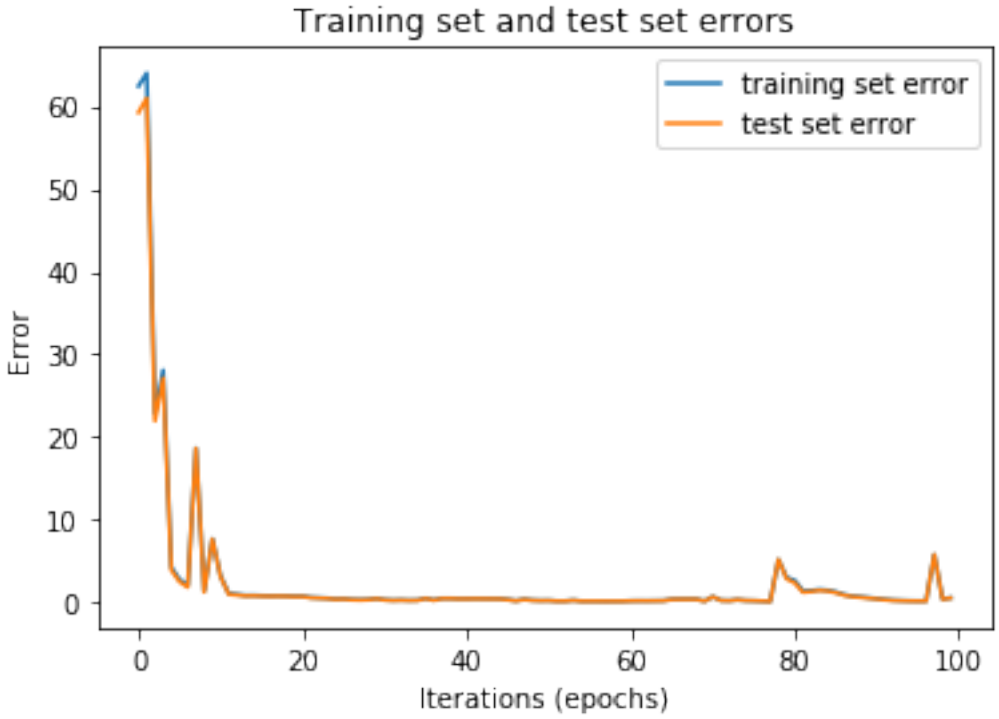

Figure 16: png

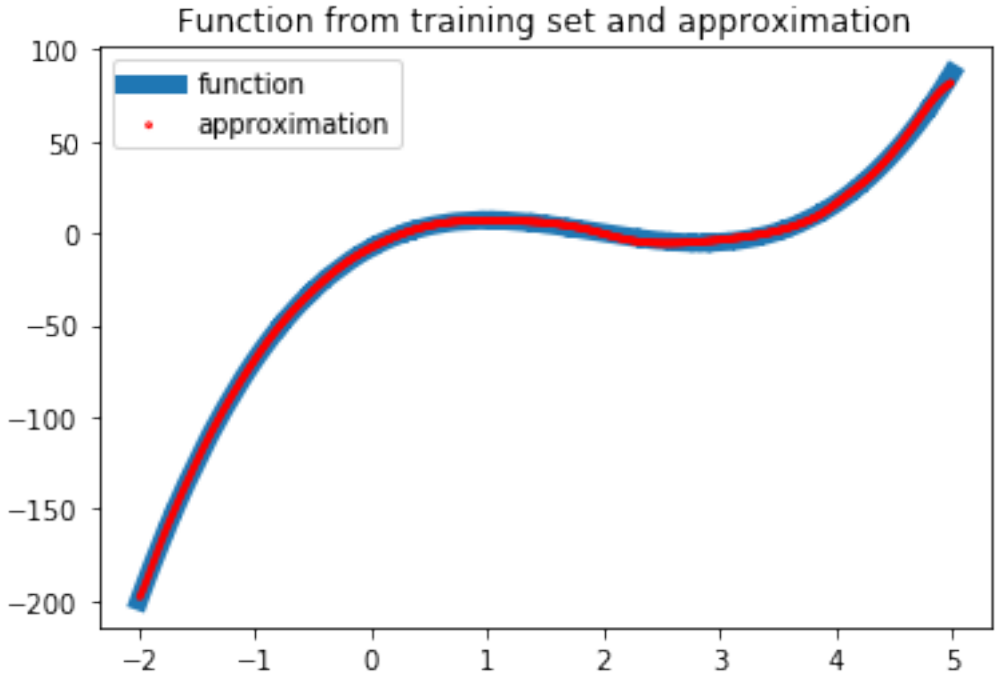

Figure 17: png

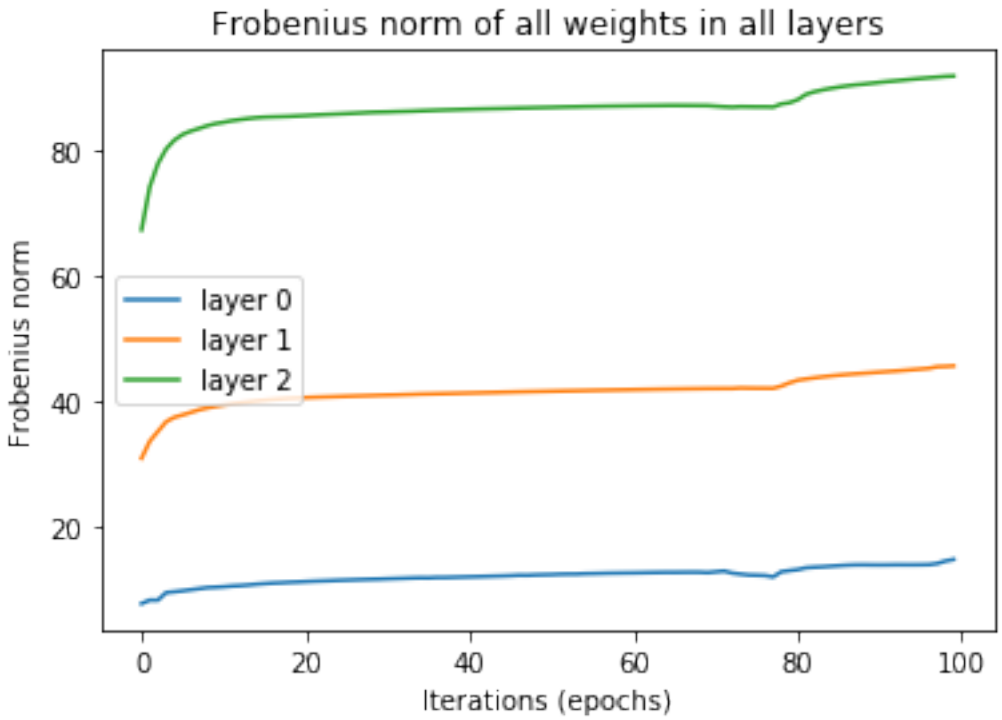

Figure 18: png

```
np.random.seed(0)
x = data_read_regression('activation',100)

# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3

brain = Network(learning_rate = 0.0001, momentum_rate = 0.7, iterations = 10000)
brain.add(Layer(1,1,'sigmoid'))
brain.add(Layer(1,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 10000/10000
```

Funkcja multimodalna

Tego typu dane sprawiają sieci neuronowej duży kłopot. Nawet złożona sieć nie jest w stanie dobrze odwzorować funkcji w początkowym i końcowym przedziale. Poniżej wykorzystamy 3-warstwową sieć o dużej ilości neuronów.

```
np.random.seed(0)
x = data_read_regression('multimodal',1000)
# Przerobienie danych w taki sposób, aby zbiór treningowy i testowy miał tą samą dziedzinę
```

Figure 19: png

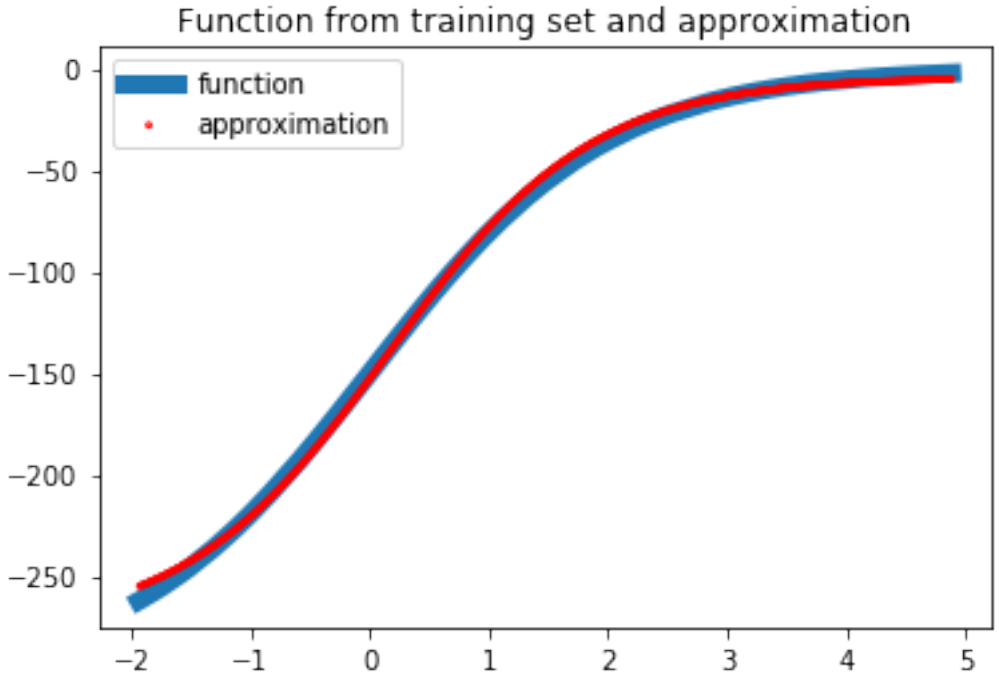

Figure 20: png

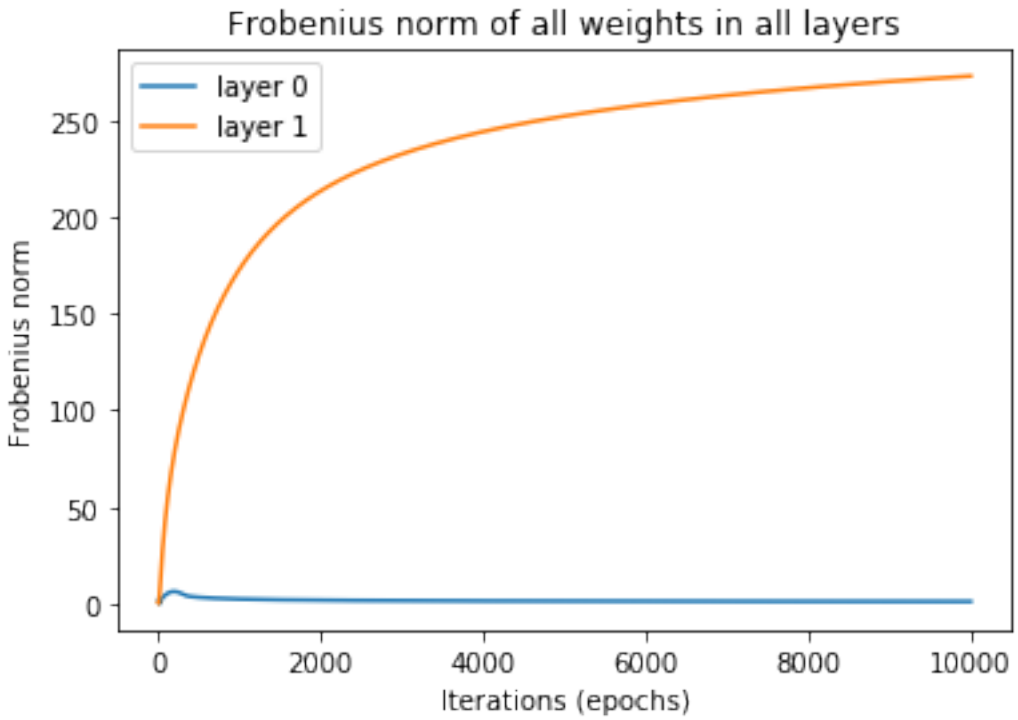

Figure 21: png

```
x3 = x[3][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x2 = x[2][(x[2] > min(x[0])) & (x[2] < max(x[0]))]
x[2] = x2
x[3] = x3

brain = Network(learning_rate = 0.001, momentum_rate = 0.1, iterations = 130)
brain.add(Layer(1,50,'sigmoid'))
brain.add(Layer(50,200,'sigmoid'))
brain.add(Layer(200,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 130/130
```

Następnie wykorzystamy jeszcze bardziej skomplikowaną sieć i jeszcze więcej danych wejściowych. Niestety widzimy że nic z tego nie będzie - funkcja jest po prostu zbyt skomplikowana.

```
np.random.seed(0)

brain = Network(learning_rate = 0.0001, momentum_rate = 0.9, iterations = 100)
brain.add(Layer(1,5,'sigmoid'))
brain.add(Layer(5,100,'sigmoid'))
brain.add(Layer(100,50,'sigmoid'))
brain.add(Layer(50,100,'sigmoid'))
brain.add(Layer(100,1,'linear'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
```

Figure 22: png

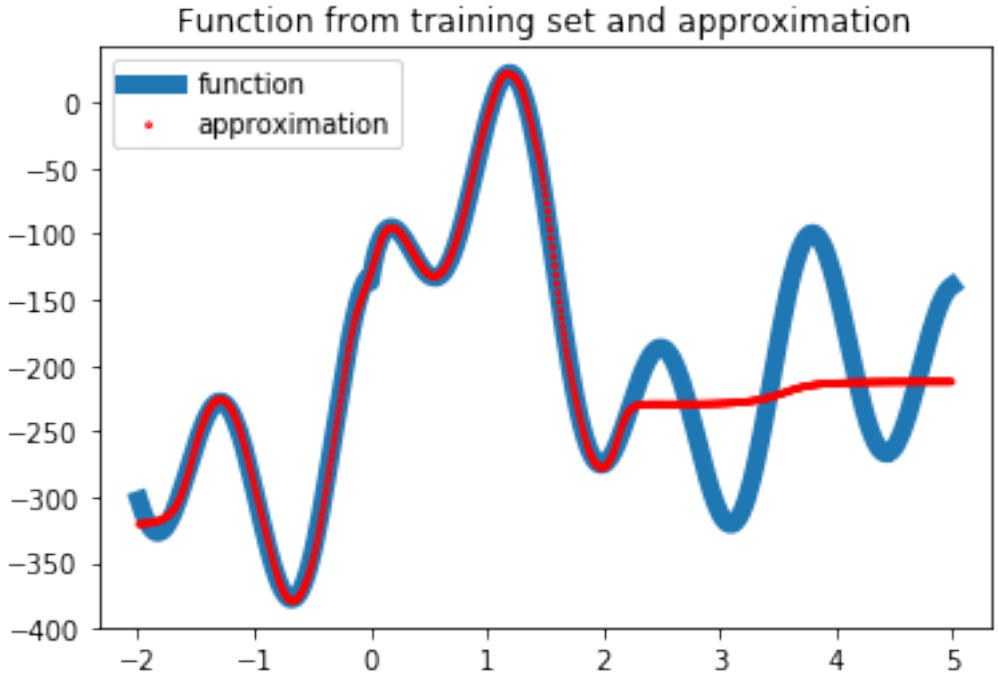

Figure 23: png

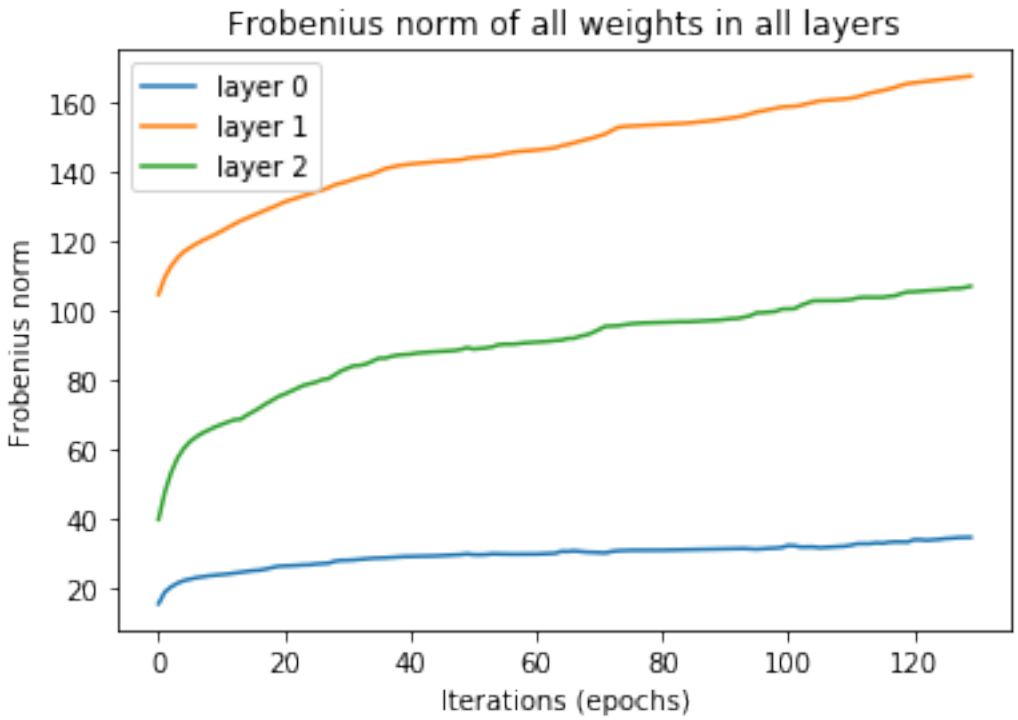

Figure 24: png

```
plot_regression(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Problem klasyfikacji

W problemie klasyfikacji na wejściu przyjmujemy dwie współrzędne, a na wyjściu powinniśmy ustawić tyle neuronów ile klas chcemy uzyskać. Wybór klasy jest determinowany przez to, który neuron w ostatniej warstwie otrzymuje największą wartość. No właśnie - wartość. Przydałoby się to w jakiś sposób znormalizować, aby wszystkie wartości w ostatniej warstwie sumowały się do 1, ponieważ tak naprawdę chcemy uzyskać prawdopodobieństwo. Z pomocą przychodzi funkcja *softmax*:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Z wykorzystaniem jej jako funkcji aktywacji w ostatniej warstwie otrzymujemy wartości sumującej się do 1.

Prosta klasyfikacja

W pierwszym przypadku zmierzymy się z prostym podziałem liniowym, na pierwszy rzut oka wydaje się, że nawet prosta sieć dwuwarstwowa powinna poradzić sobie z tym problemem. Zgodnie z oczekiwaniami, wystarczy bardzo prosta sieć z jednym neuronem w pierwszej warstwie aby rozwiązać liniowy problem klasyfikacji.

```
np.random.seed(0)
x = data_read_classification('simple',10000)
```

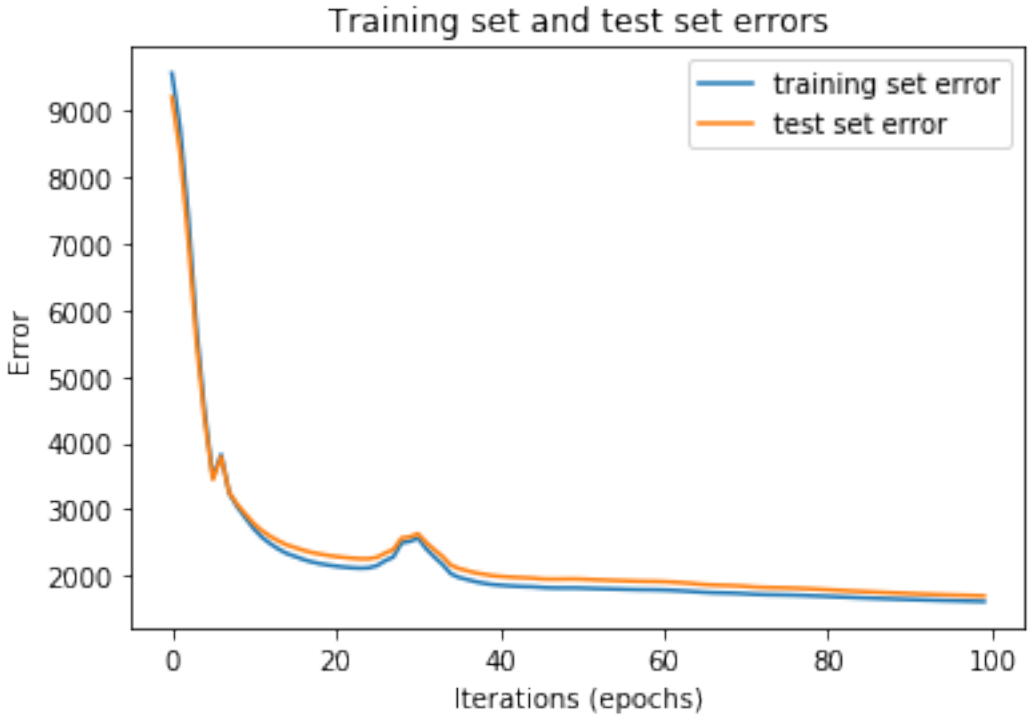

Figure 25: png

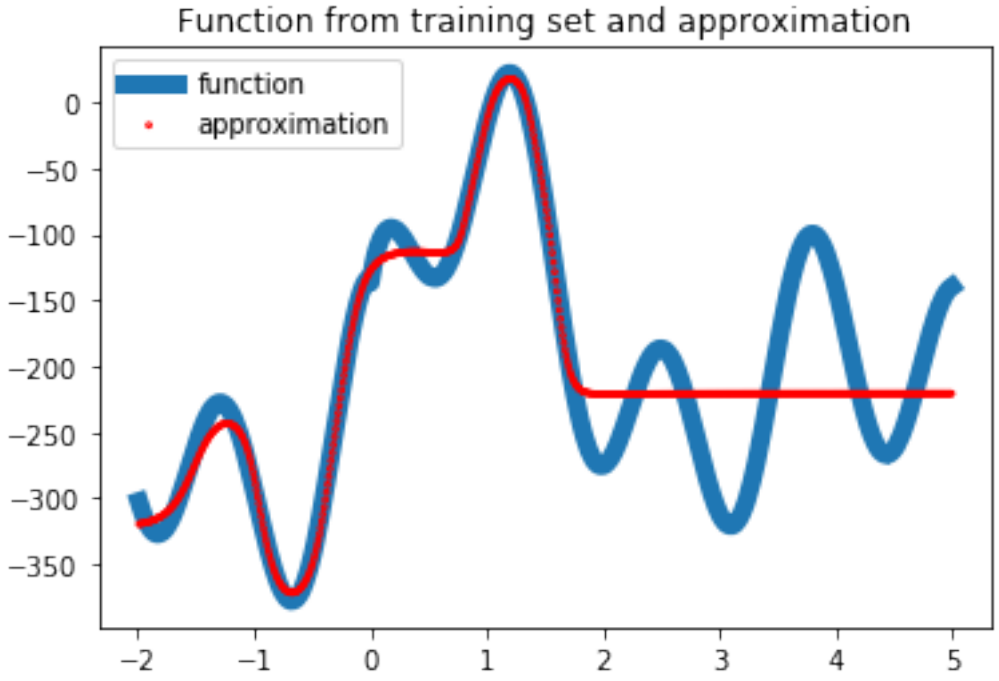

Figure 26: png

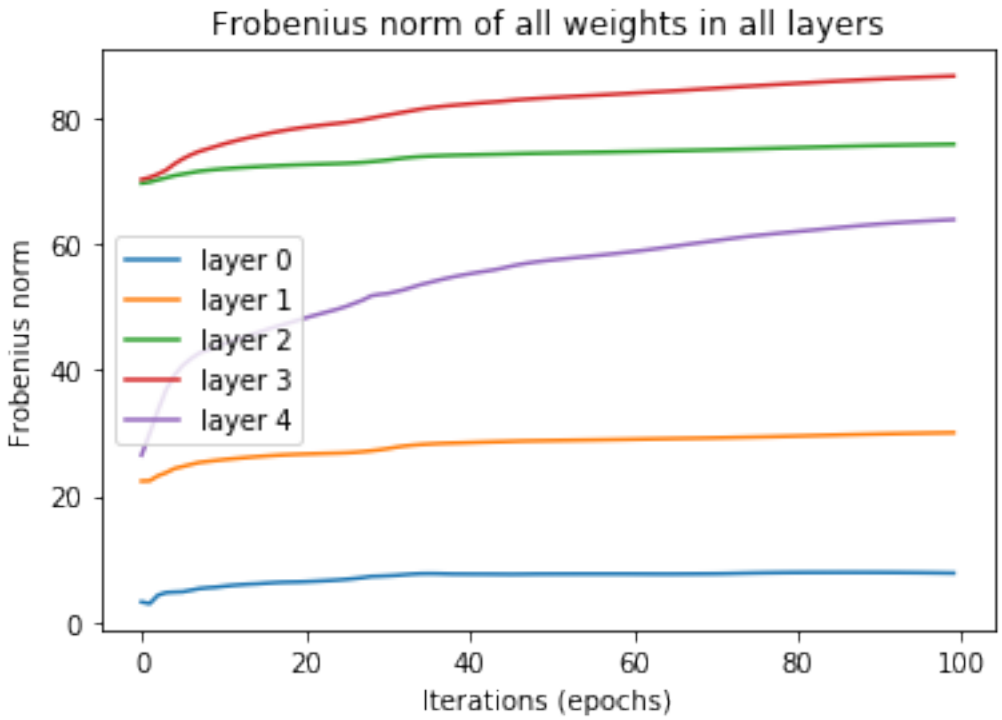

Figure 27: png

```
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 100)
brain.add(Layer(2,1,'sigmoid'))
brain.add(Layer(1,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
#errors = brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

XOR

XOR jest już problemem, którego nie da się rozwiązać liniowo, ponieważ musielibyśmy narysować minimum dwie linie, żeby oddzielić segmenty w wynikach. Dlatego w tym przypadku zwiększymy rozmiar pierwszej warstwy do 4 neuronów

Próba liniowego rozwiązania XOR

```
np.random.seed(0)
x = data_read_classification('xor',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.9, iterations = 100)
brain.add(Layer(2,1,'sigmoid'))
brain.add(Layer(1,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
#errors = brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)
plot_errors(errors)
```

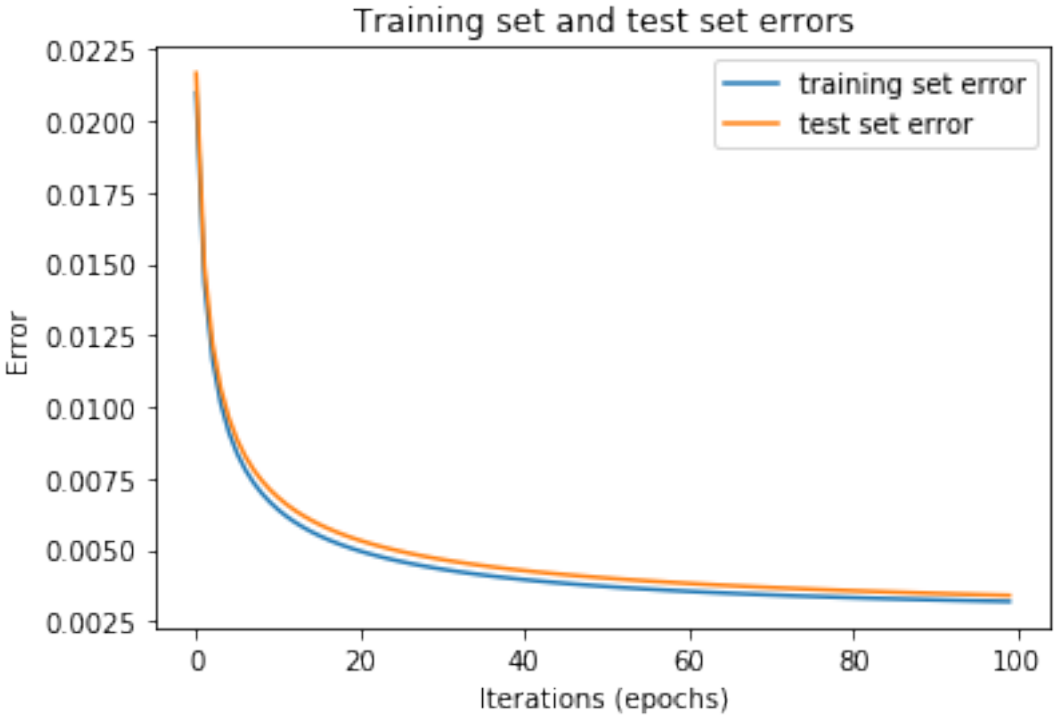

Figure 28: png

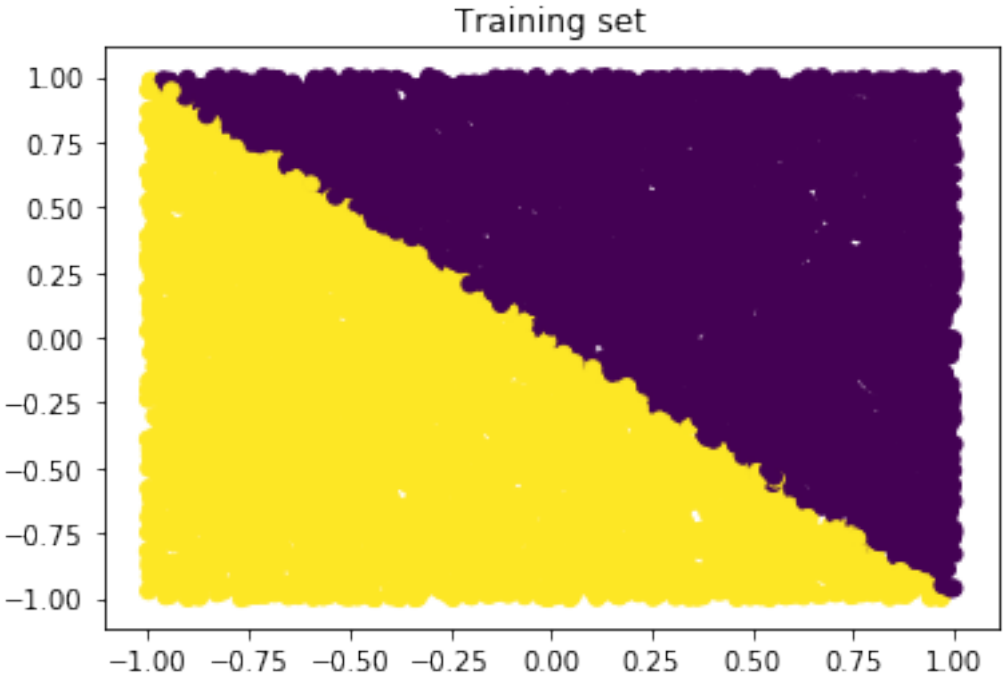

Figure 29: png

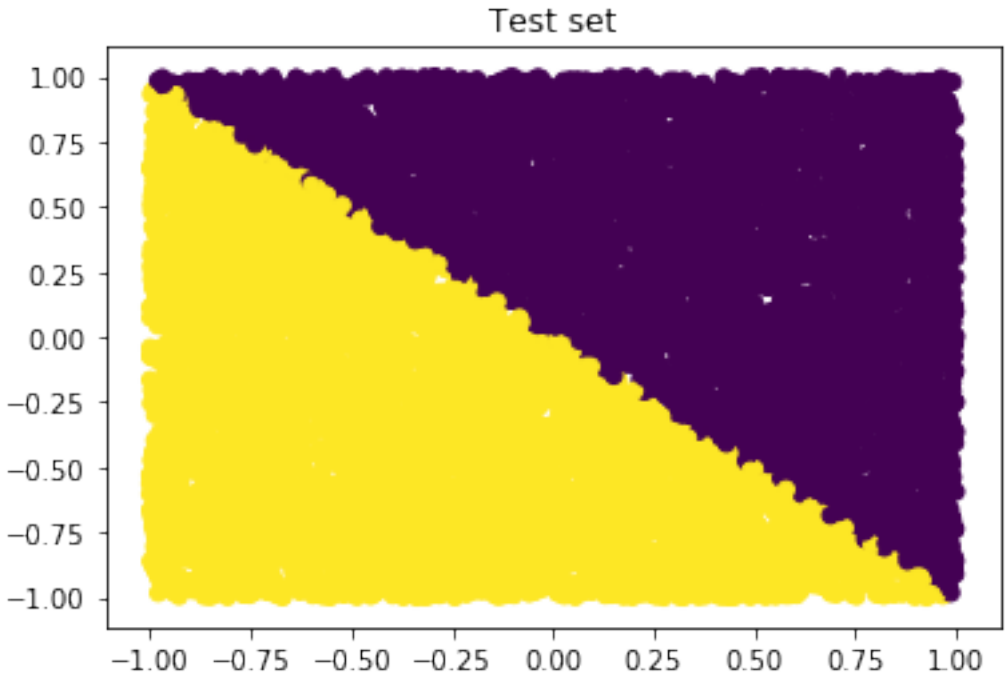

Figure 30: png

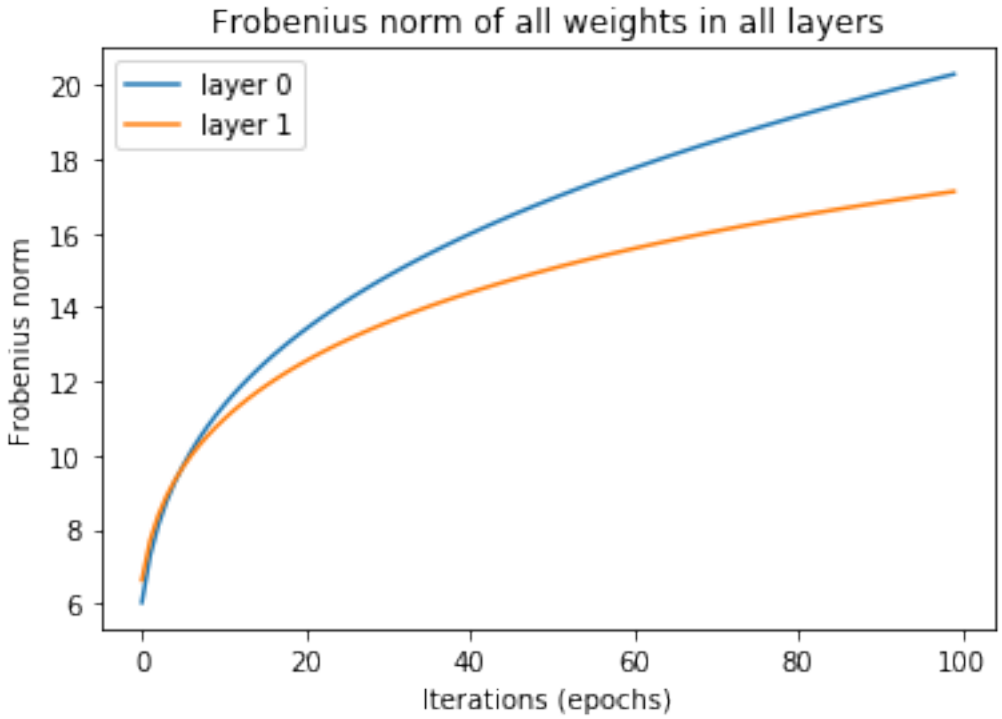

Figure 31: png

```
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

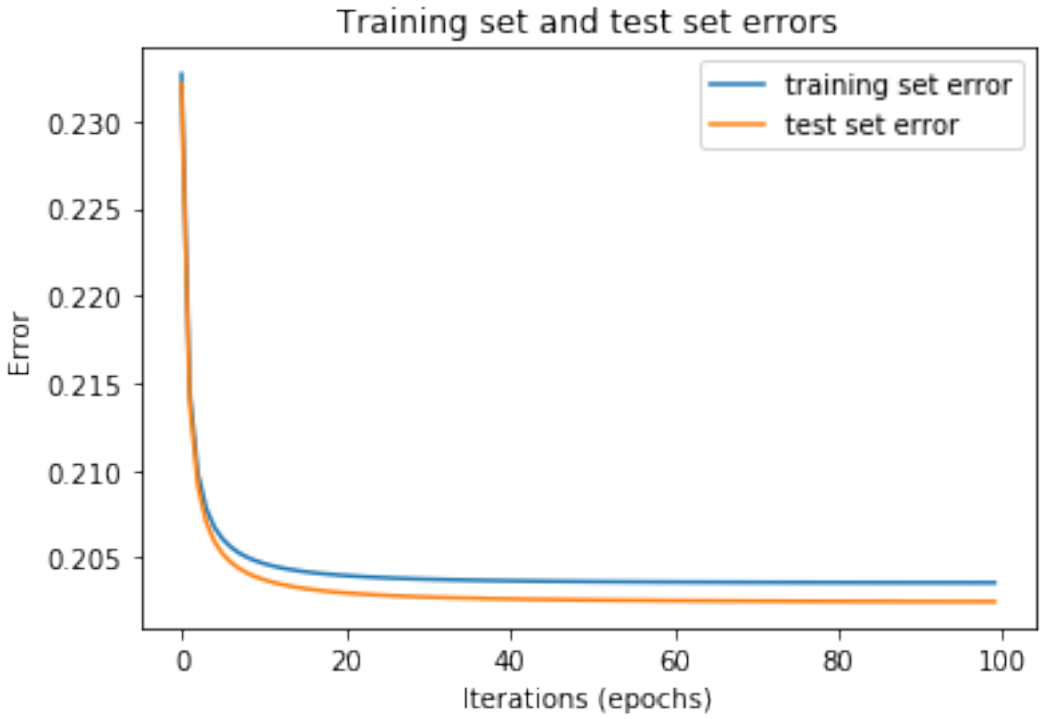

Figure 32: png

Rozwiązanie dla XOR z 4 neuronami w pierwszej warstwie

```
np.random.seed(0)
x = data_read_classification('xor',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.9, iterations = 100)
brain.add(Layer(2,4,'sigmoid'))
brain.add(Layer(4,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
#errors = brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Noisy XOR

Jest to podobny problem jak w przypadku zwykłego XOR, jednak dane są o wiele bardziej zaszumione.

Na początku użyjemy podobnego ustawienia parametrów sieci jak w przypadku niezaszumionego XOR, a w drugim przykładzie wykorzystamy o wiele bardziej skomplikowaną sieć i porównamy błędy.

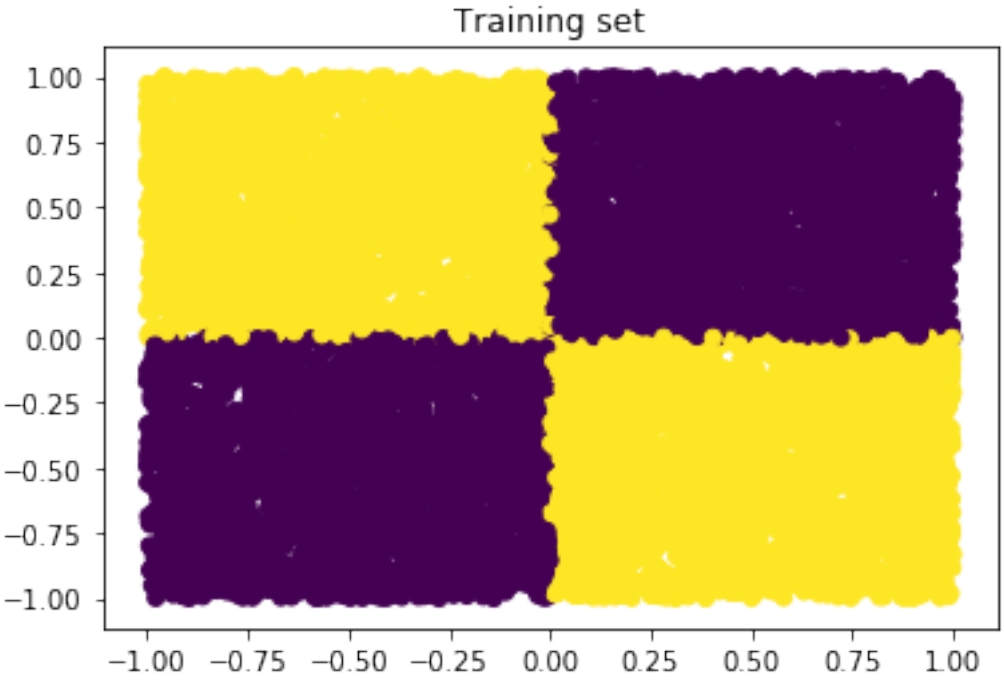

Figure 33: png

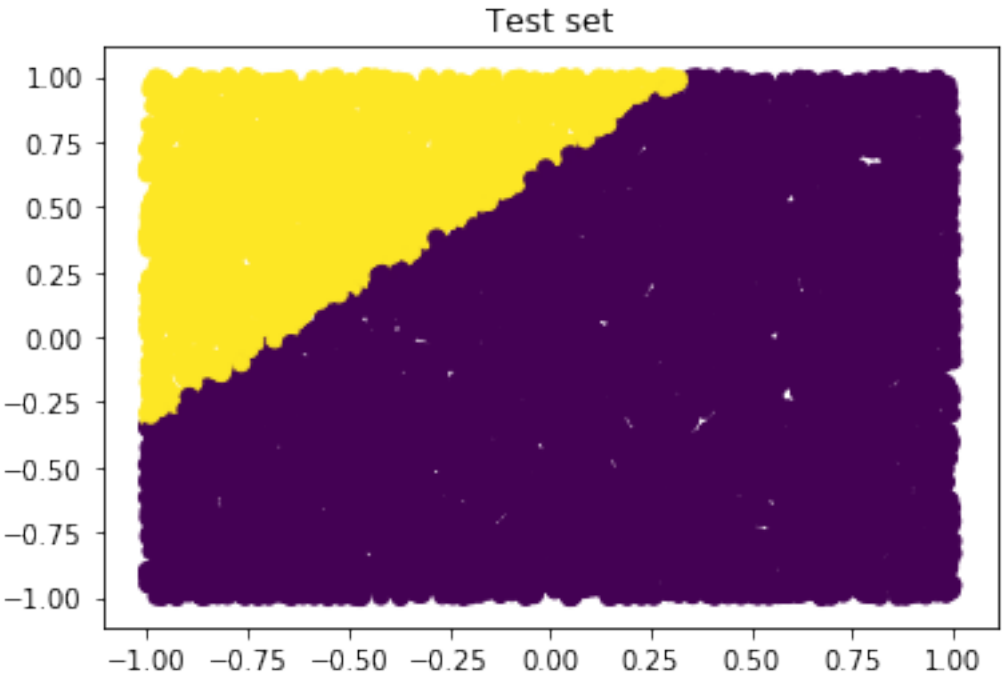

Figure 34: png

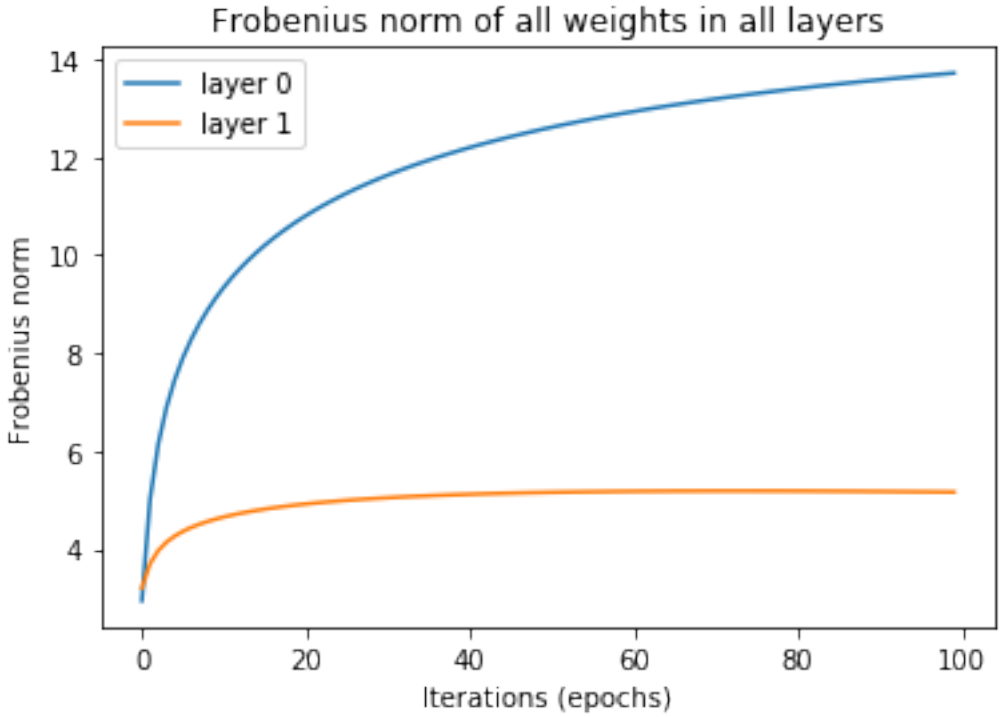

Figure 35: png

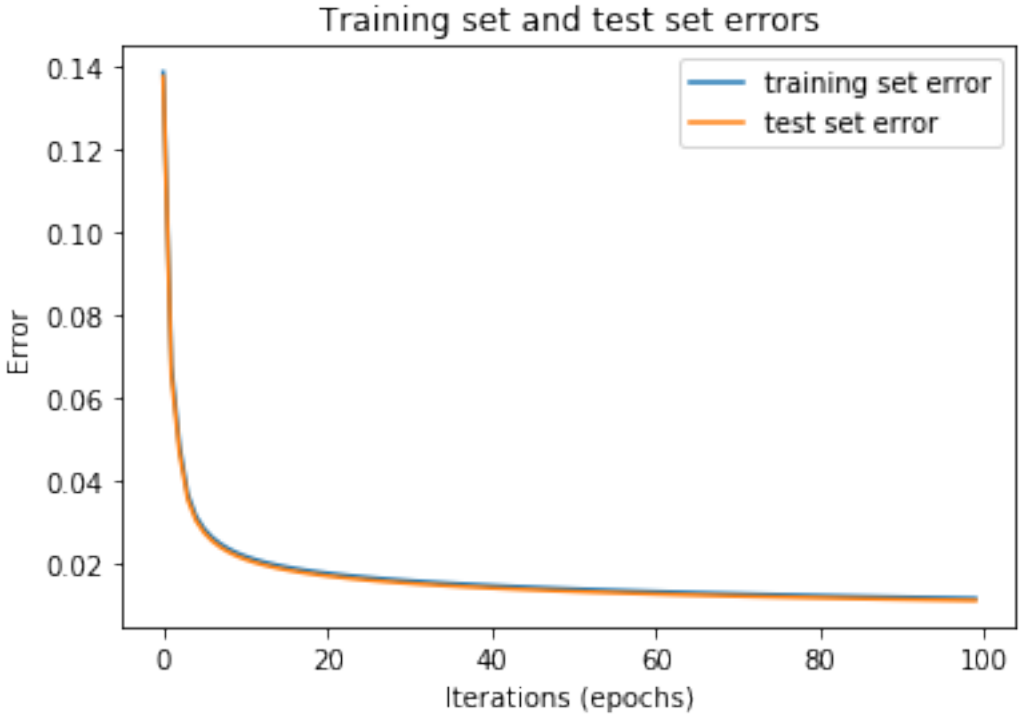

Figure 36: png

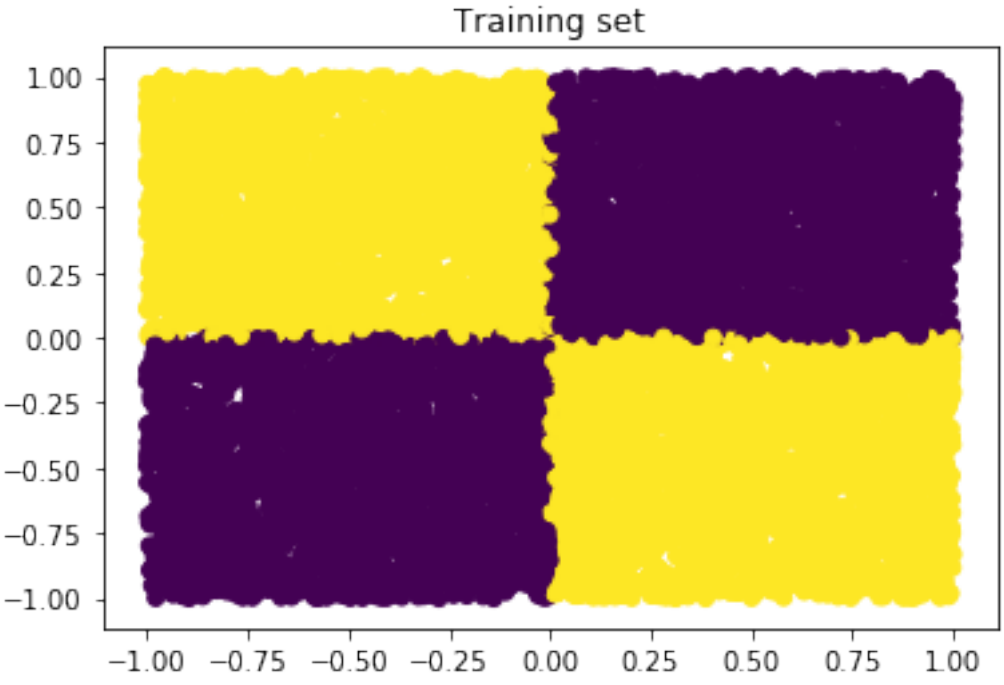

Figure 37: png

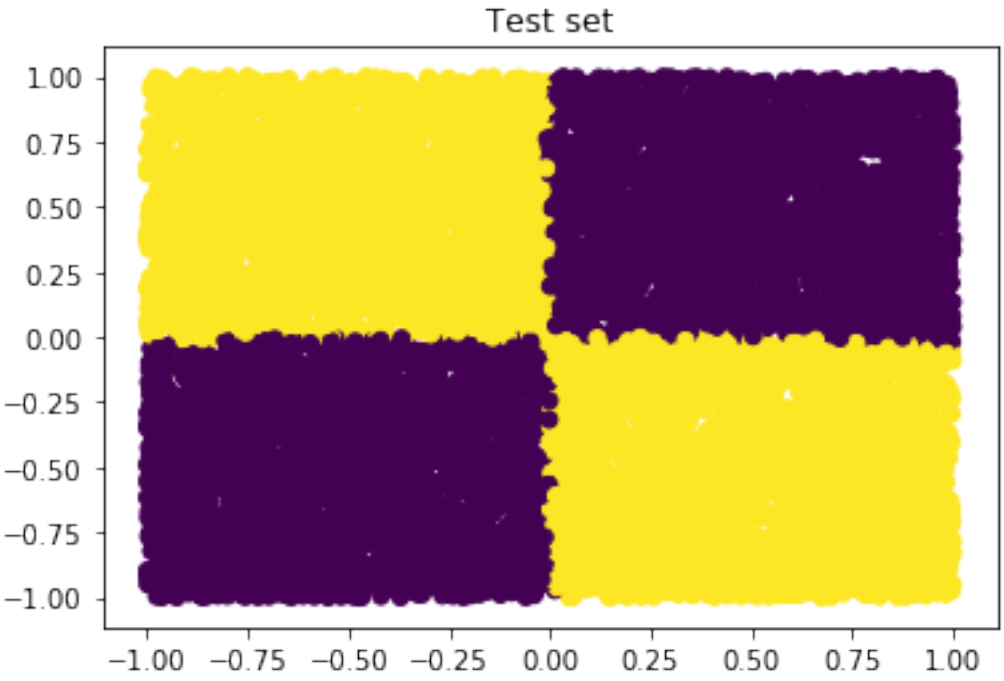

Figure 38: png

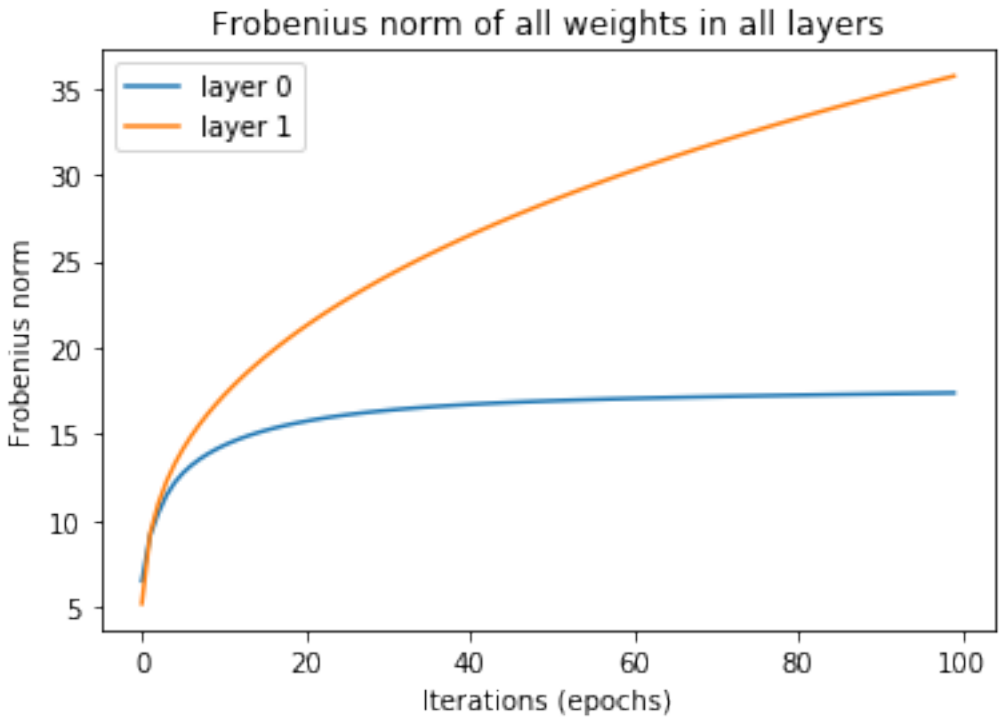

Figure 39: png

Takie same ustawienia jak w przypadku niezaszumionego XOR

Dane są dosyć mocno zniekształcone, więc mozemy się spodziewać że zaszumiony obszar zostanie po prostu przypisany do jednej z dwóch klas.

```
np.random.seed(0)
x = data_read_classification('noisyxor',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.9, iterations = 100)
brain.add(Layer(2,4,'sigmoid'))
brain.add(Layer(4,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
#errors = brain.train_mini_batch_and_evaluate(x[0],x[1],x[2],x[3],10)
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Podobne ustawienia jak w przypadku niezaszumionego XOR, ale z większą liczbą warstw ukrytych

Możemy zauważyć że powiększanie sieci w przypadku zaszumionych danych nie daje lepszych rezultatów. Sieć nie umie sobie poradzić z takim typem danych.

```
np.random.seed(0)
x = data_read_classification('noisyxor',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 80)
brain.add(Layer(2,20,'sigmoid'))
brain.add(Layer(20,50,'sigmoid'))
```

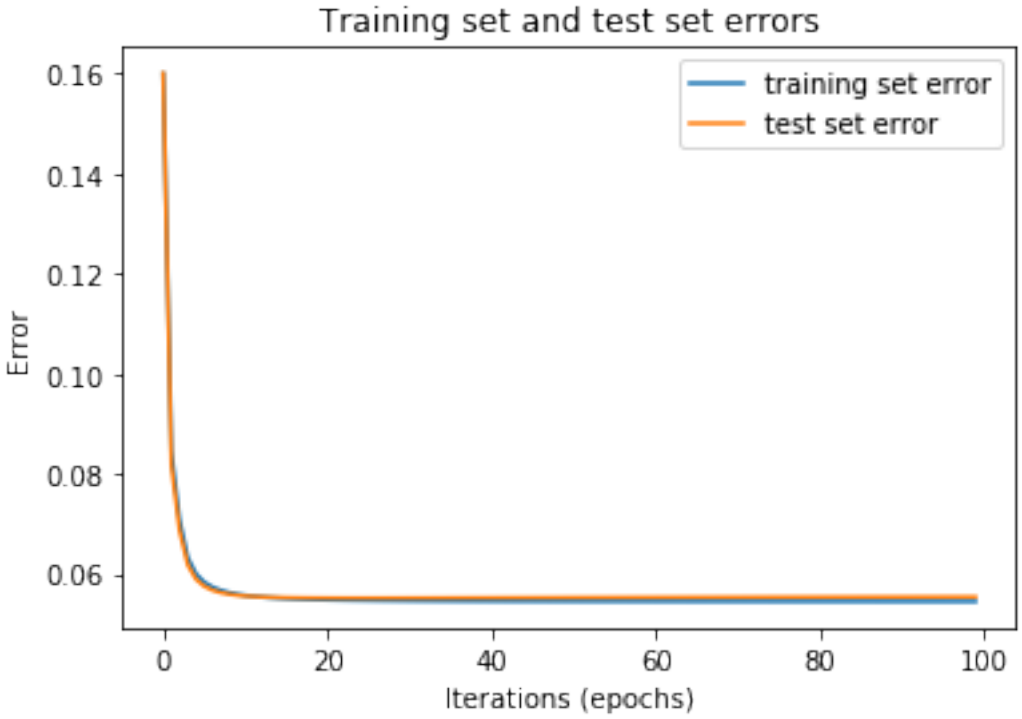

Figure 40: png

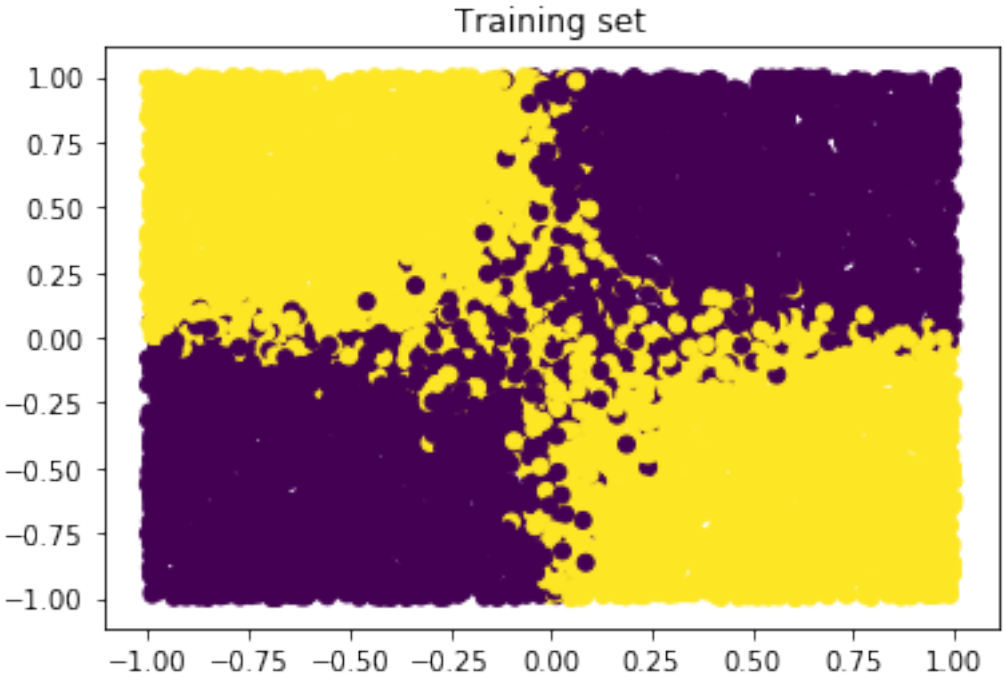

Figure 41: png

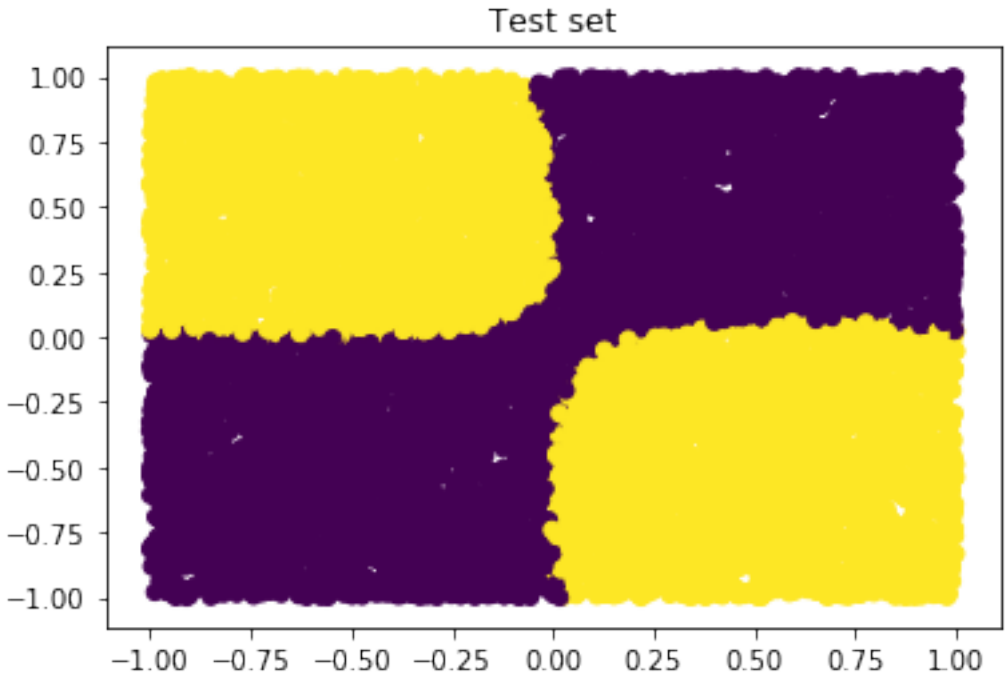

Figure 42: png

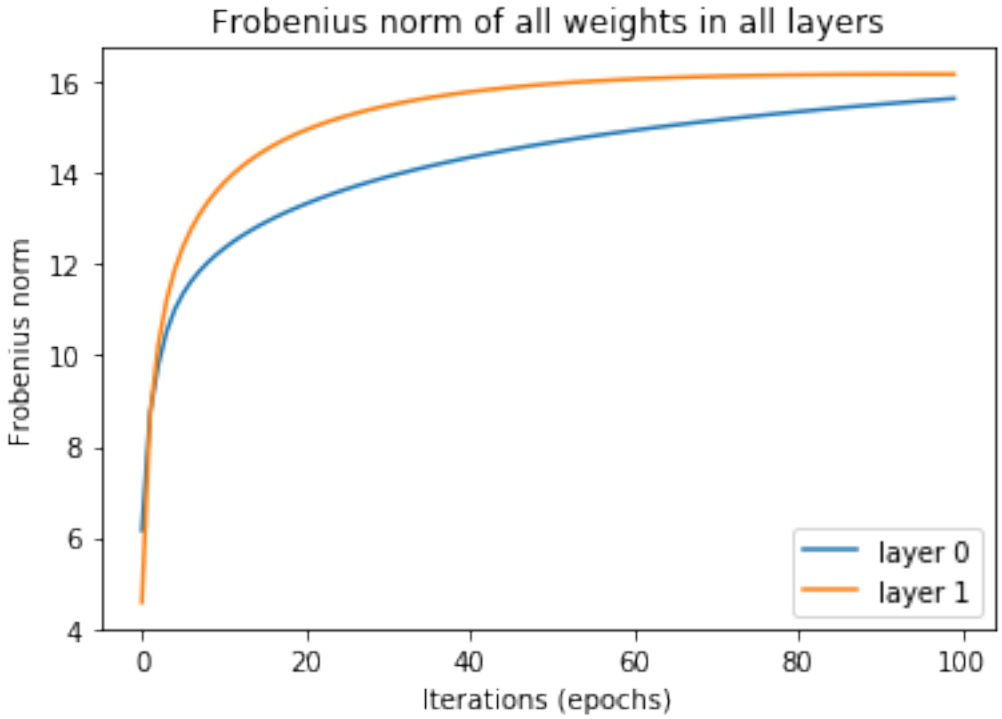

Figure 43: png

```
brain.add(Layer(50,10,'sigmoid'))
brain.add(Layer(10,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 80/80
```

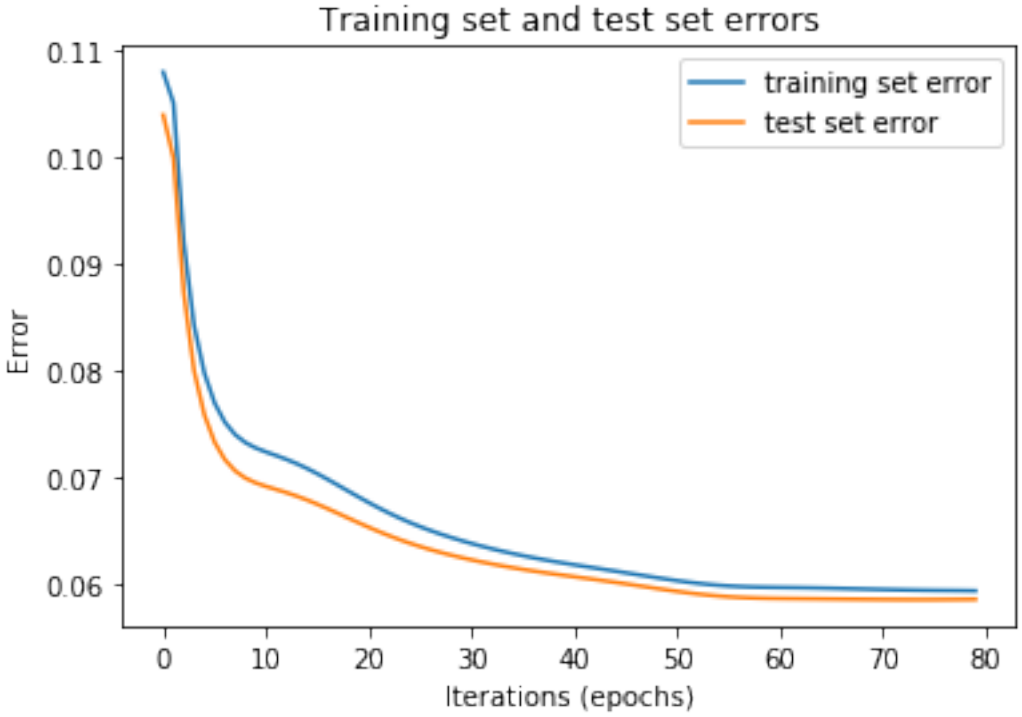

Figure 44: png

Zobaczmy jeszcze w jaki sposób wygląda dopasowanie na troche innym wykresie (musieliśmy użyć trochę mniejszego zbioru danych, żeby było cokolwiek widać): - mniej intensywne kolory to zbiór testowy (obszary klasyfikacyjne) - bardziej intensywne kolory to zbiór treningowy

Tam gdzie mniej intensywny kolor nie zgadza się z bardziej intensywnym znaczy że dopasowanie nie zostało dobrze przeprowadzone.

Niestety na tych kilku przykładach widać, że sieć nie potrafi sobie poradzić z zaszumionym obszarem, zostaje on przypisany albo do jednej albo drugiej klasy.

```
np.random.seed(0)
x = data_read_classification('noisyxor',1000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 80)
brain.add(Layer(2,20,'sigmoid'))
brain.add(Layer(20,50,'sigmoid'))
brain.add(Layer(50,10,'sigmoid'))
brain.add(Layer(10,2,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_classification_mesh(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 80/80
```

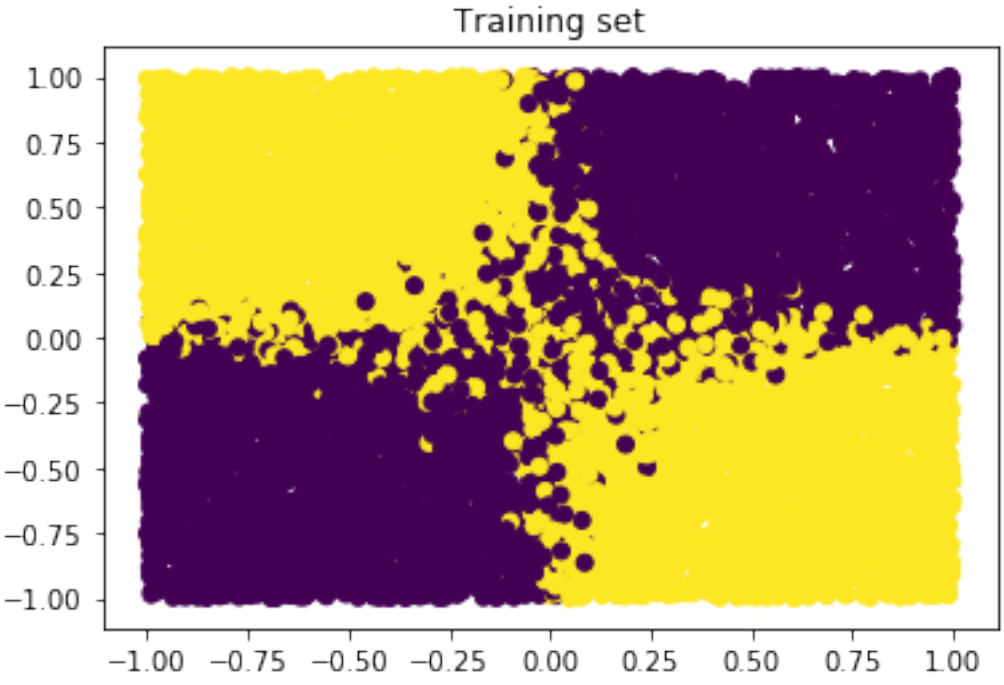

Figure 45: png

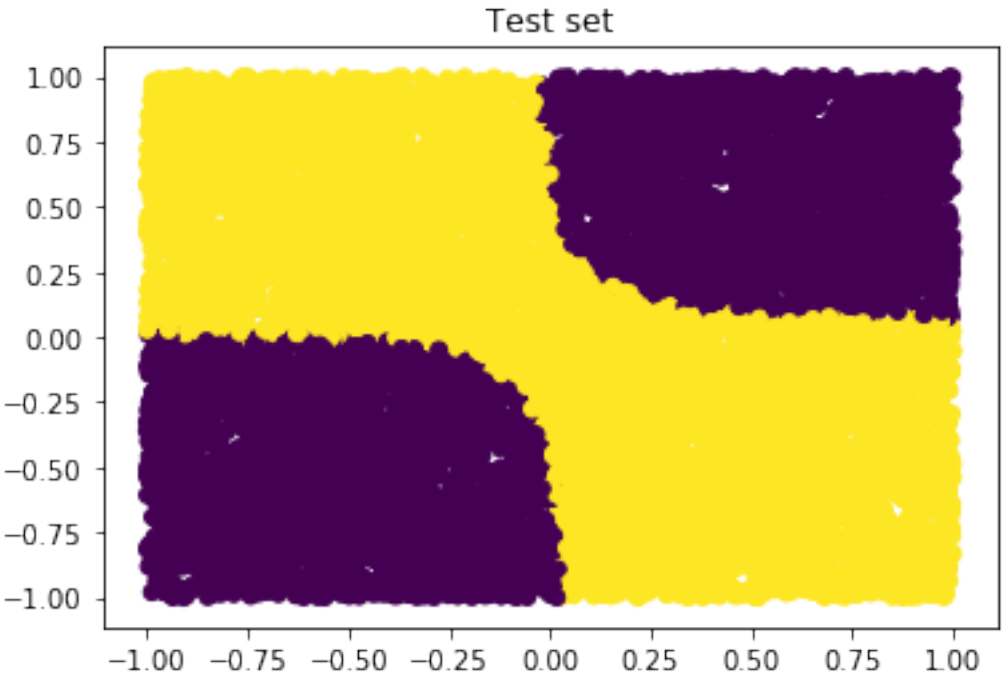

Figure 46: png

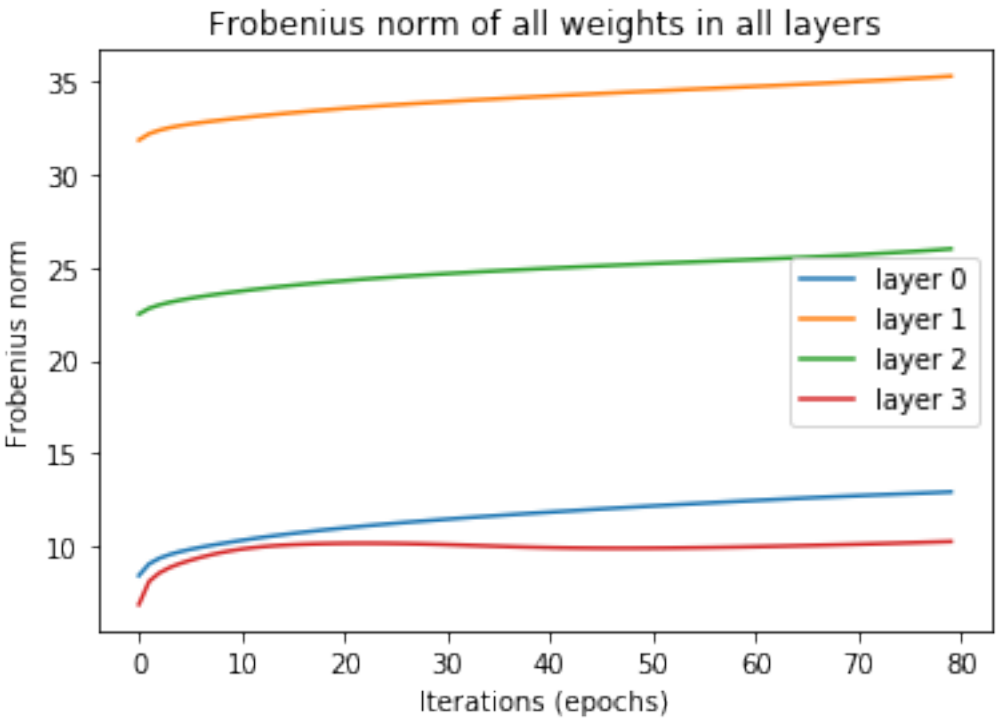

Figure 47: png

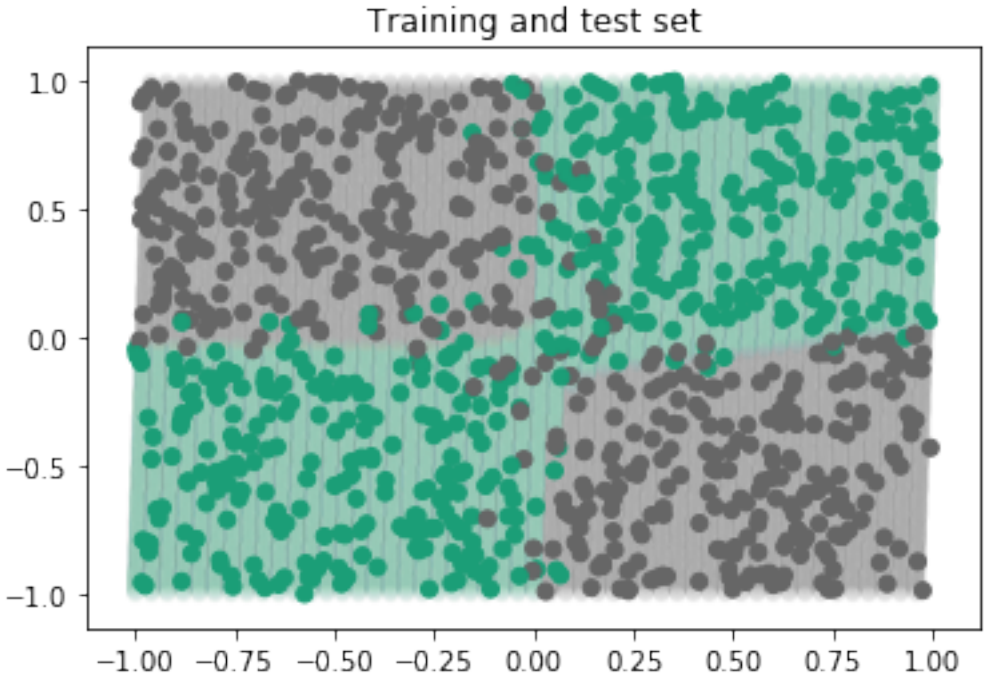

Figure 48: png

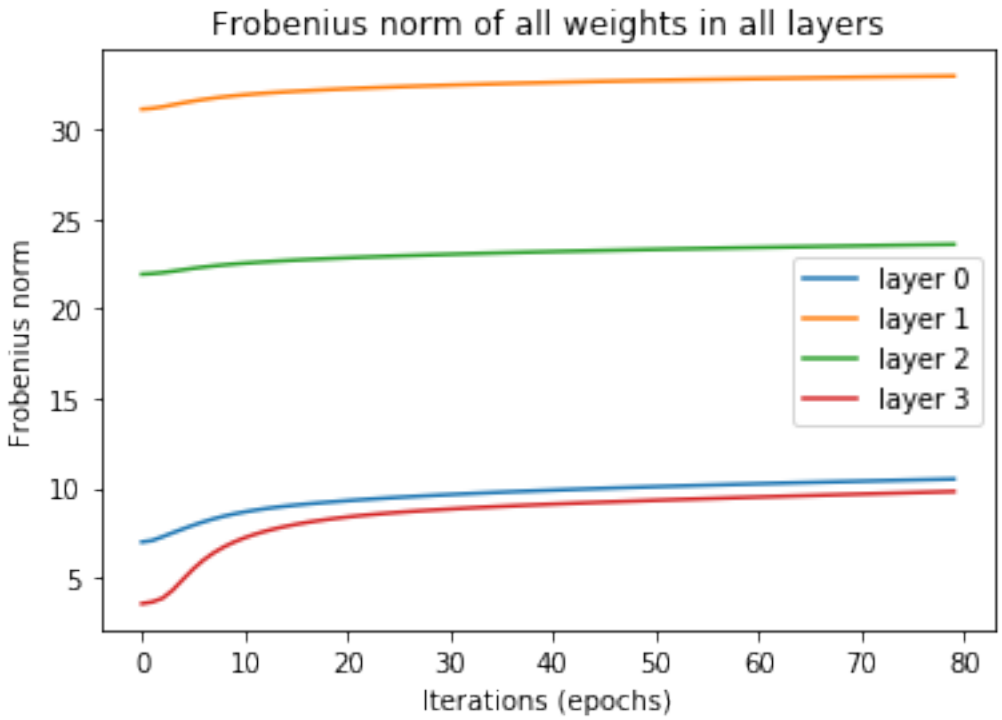

Figure 49: png

Three Gauss

Jest to dosyć skomplikowany problem, ponieważ 3 klasy nachodzą na siebie w dosyć dużym stopniu. Na początek zobaczmy jak wygląda dopasowanie dla małego zbioru danych. Możemy zauważyć, że sieć podzieliła przestrzeń na 3 segmenty i w taki sposób grupuje nam dane wejściowe. Podobnie obszar zaszumiony jest stopniowo podzielony między 3 klasy. Ostatecznie wydaje się, że jest to dobre rozwiązanie.

```
np.random.seed(0)
x = data_read_classification('three_gauss',100)
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 100)
brain.add(Layer(2,20,'sigmoid'))
brain.add(Layer(20,30,'sigmoid'))
brain.add(Layer(30,3,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification_mesh(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Zobaczmy teraz co dzieje się dla większego zbioru danych. Zgodnie z oczekiwaniami otrzymujemy podobny podział i skuteczność.

```
np.random.seed(0)
x = data_read_classification('three_gauss',1000)
brain = Network(learning_rate = 0.01, momentum_rate = 0.8, iterations = 100)
brain.add(Layer(2,20,'sigmoid'))
brain.add(Layer(20,30,'sigmoid'))
brain.add(Layer(30,3,'softmax'))
```

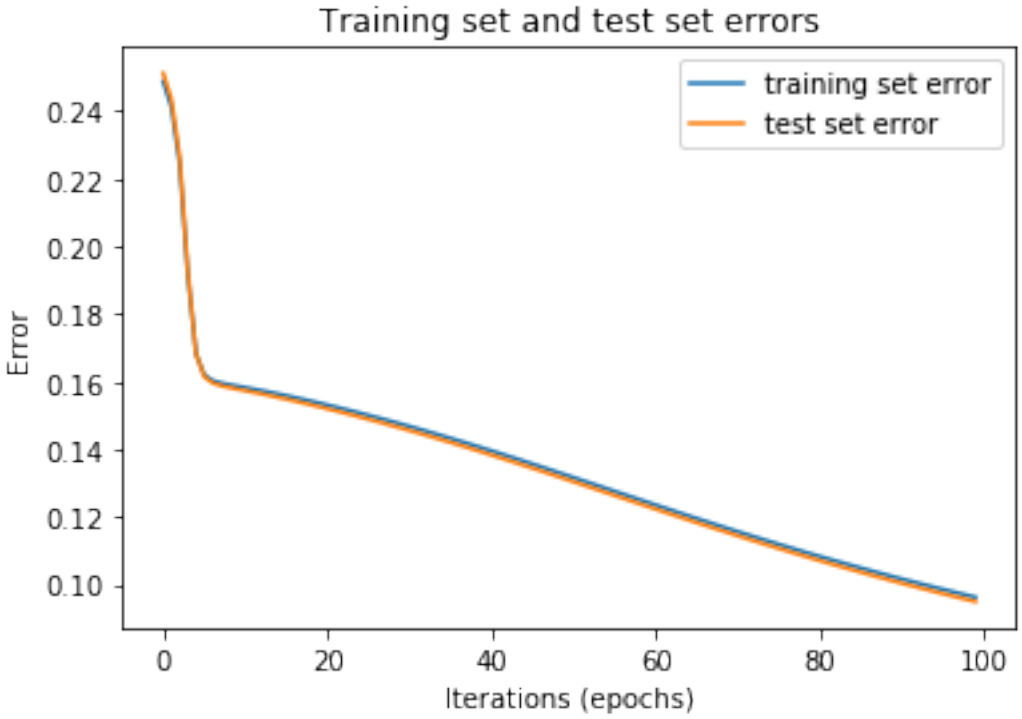

Figure 50: png

Figure 51: png

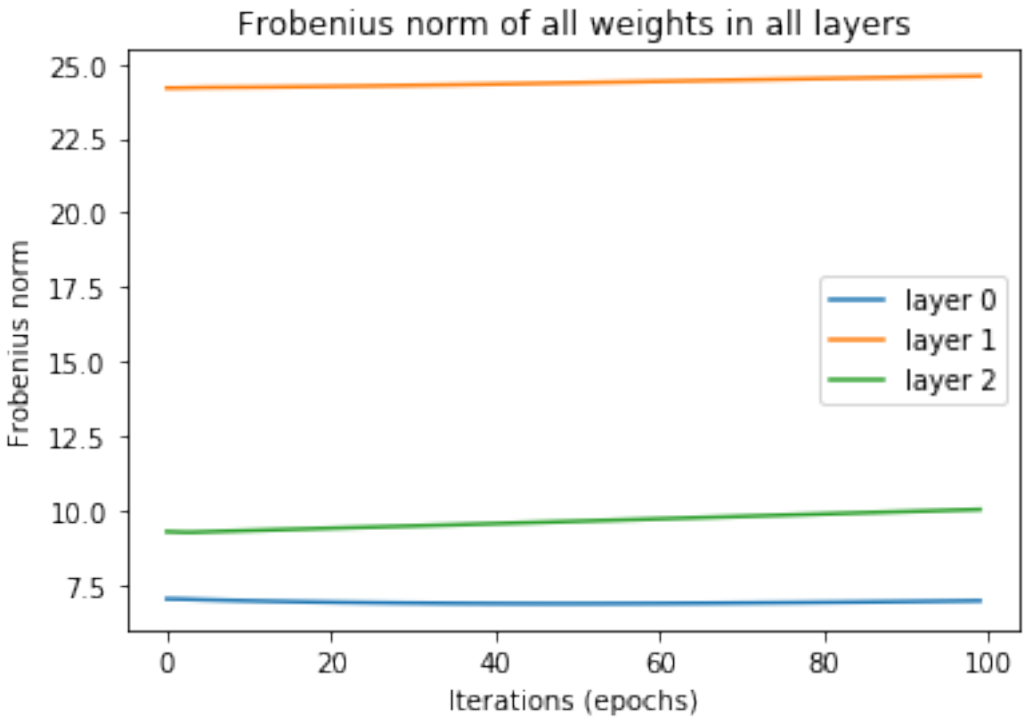

Figure 52: png

```
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification_mesh(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 100/100
```

Circles

Jest to najbardziej skomplikowany zbiór do klasyfikacji. Na obrazku występuje dużo różnych wzorców. Jednak nasza sieć z odpowiednią ilośćią warstw i neuronów w nich zawartych powinna poradzić sobie z tym problemem z dosyć dużą skutecznością.

Na sam początek zobaczmy co dzieje się dla małego zbioru danych. Widać ewidentnie, że punktów jest trochę za mało aby wykryć jakiekolwiek grupy, które mają odzwierciedlenie na zbiorze testowym, nie pomaga nawet zwiększenie liczby epok na których uczy się sieć. Funkcja błędu w pewnym momencie wypłaszcza się i zamiast delikatnie maleć - rośnie.

```
np.random.seed(0)
x = data_read_classification('circles',100)
brain = Network(learning_rate = 0.01, momentum_rate = 0.8, iterations = 1000)
brain.add(Layer(2,10,'sigmoid'))
brain.add(Layer(10,50,'sigmoid'))
brain.add(Layer(50,25,'sigmoid'))
brain.add(Layer(25,4,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification_mesh(brain, x)
weights_norms_plot(errors)
```

Figure 53: png

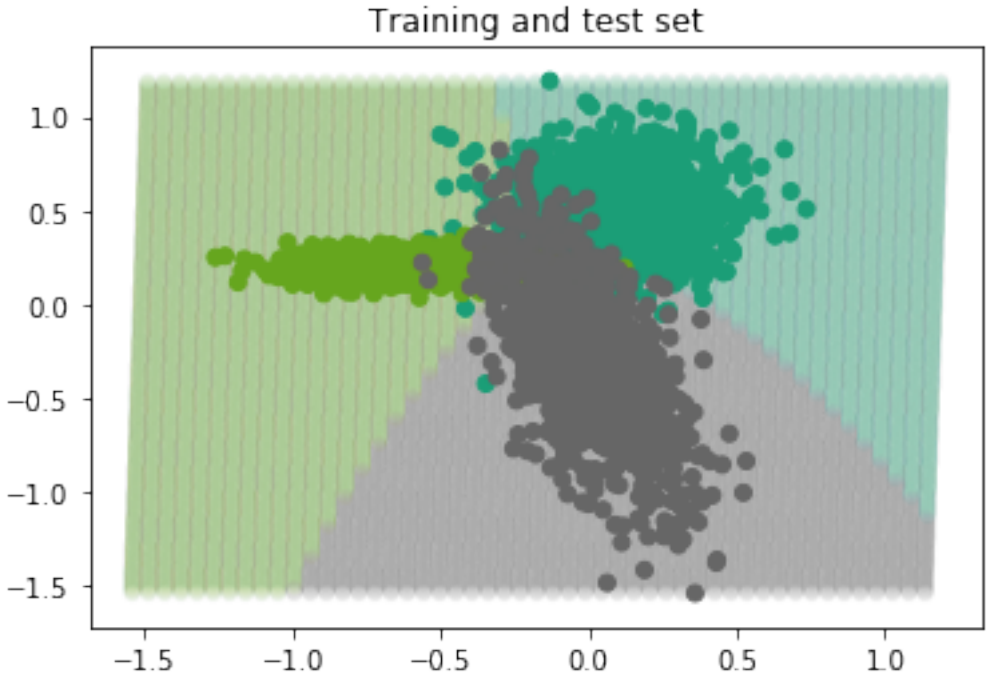

Figure 54: png

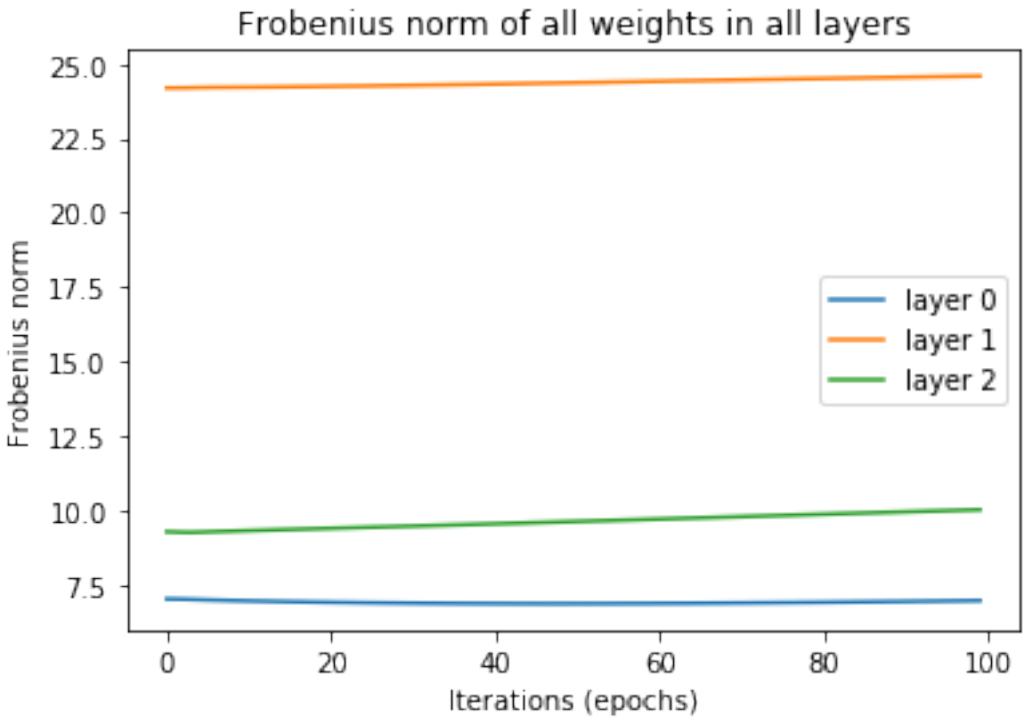

Figure 55: png

```
Epoch: 1000/1000
```

Po zwiększeniu wielkości zbioru treningowego możemy zauważyć, że sieć dosyć dobrze klasyfikuje punkty na obrzeżach obrazka. Natomiast wydaje się mieć dosyć duży problem z obszarem w centrum obrazka, gdzie dwie grupy punktów dosyć mocno na siebie nachodzą.

```
np.random.seed(0)
x = data_read_classification('circles',1000)
brain = Network(learning_rate = 0.01, momentum_rate = 0.8, iterations = 1000)
brain.add(Layer(2,10,'sigmoid'))
brain.add(Layer(10,50,'sigmoid'))
brain.add(Layer(50,25,'sigmoid'))
brain.add(Layer(25,4,'softmax'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification_mesh(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 1000/1000
```

Zwiększmy jeszcze zbiór treningowy do 10000.

Możemy zauważyć że przy odpowiedniej liczbie warstw, neuronów i danych treningowych sieć poradzi sobie nawet z najtrudniejszym przypadkiem klasyfikacji

```
np.random.seed(1)
x = data_read_classification('circles',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 20)
brain.add(Layer(2,64,'sigmoid'))
brain.add(Layer(64,256,'sigmoid'))
```

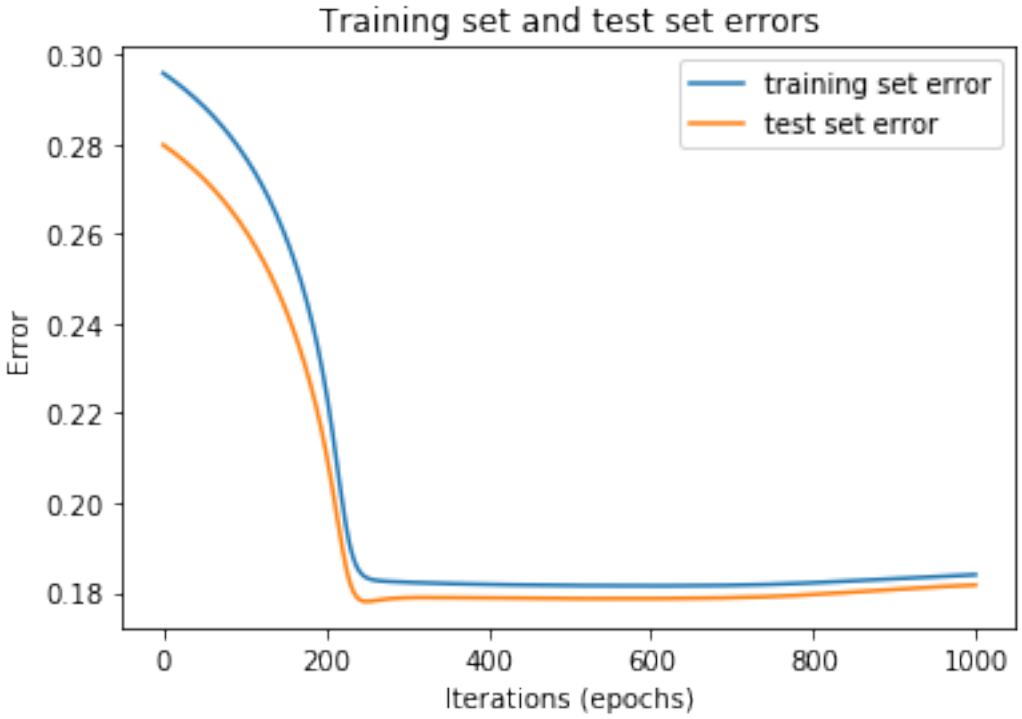

Figure 56: png

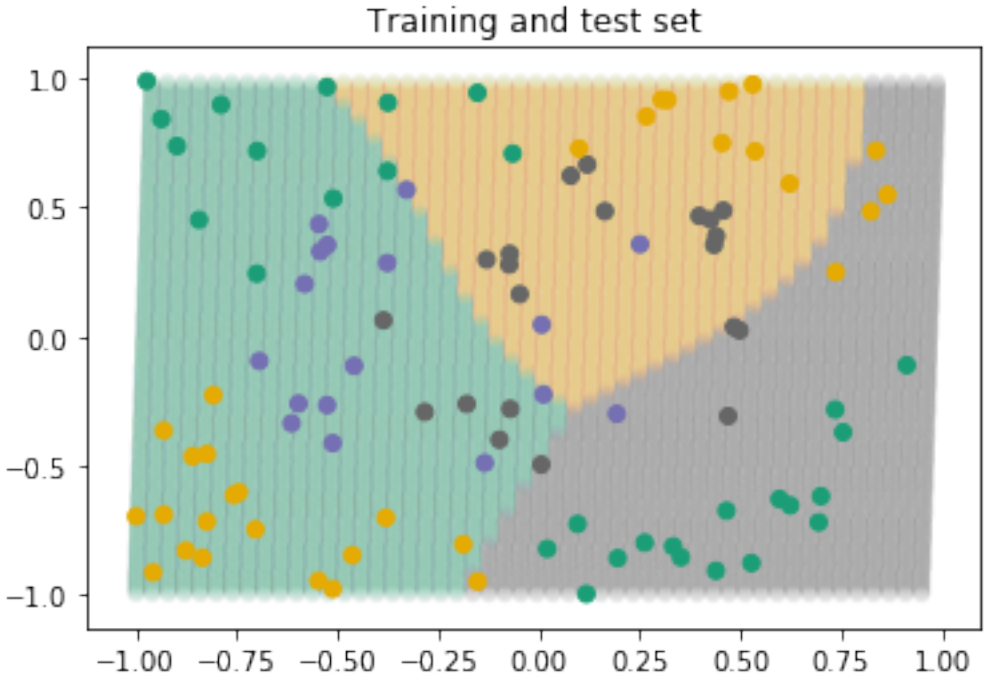

Figure 57: png

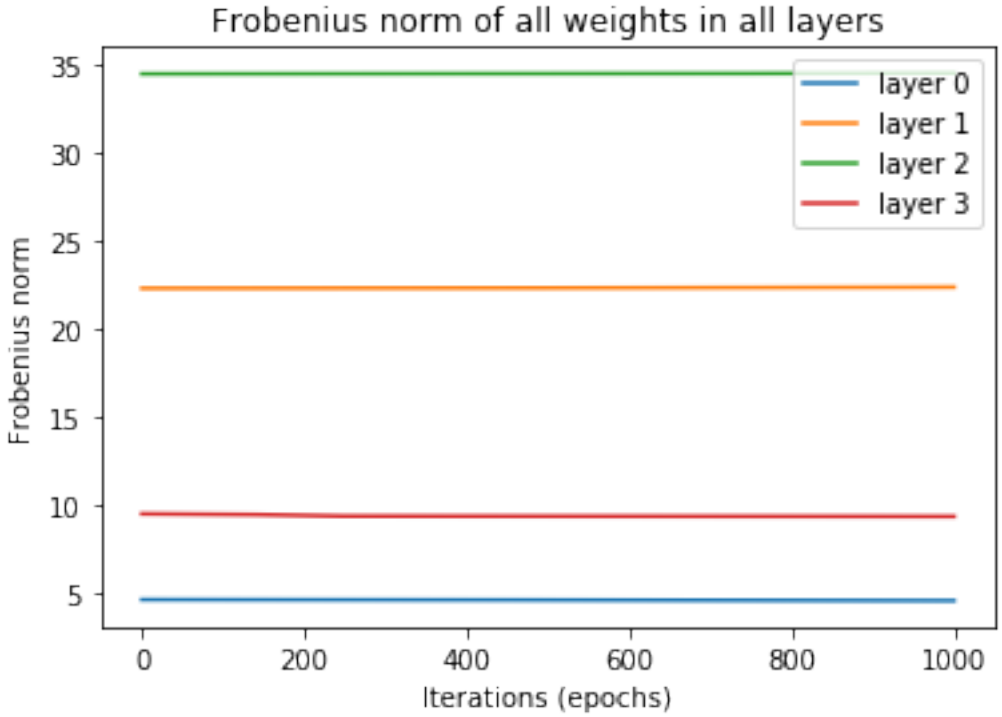

Figure 58: png

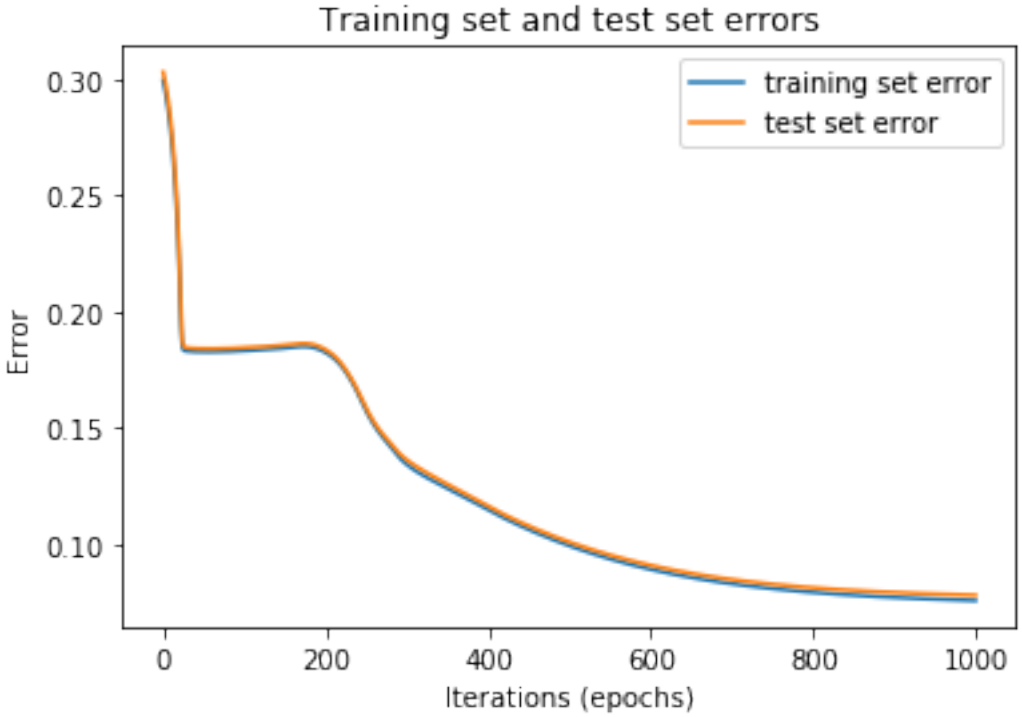

Figure 59: png

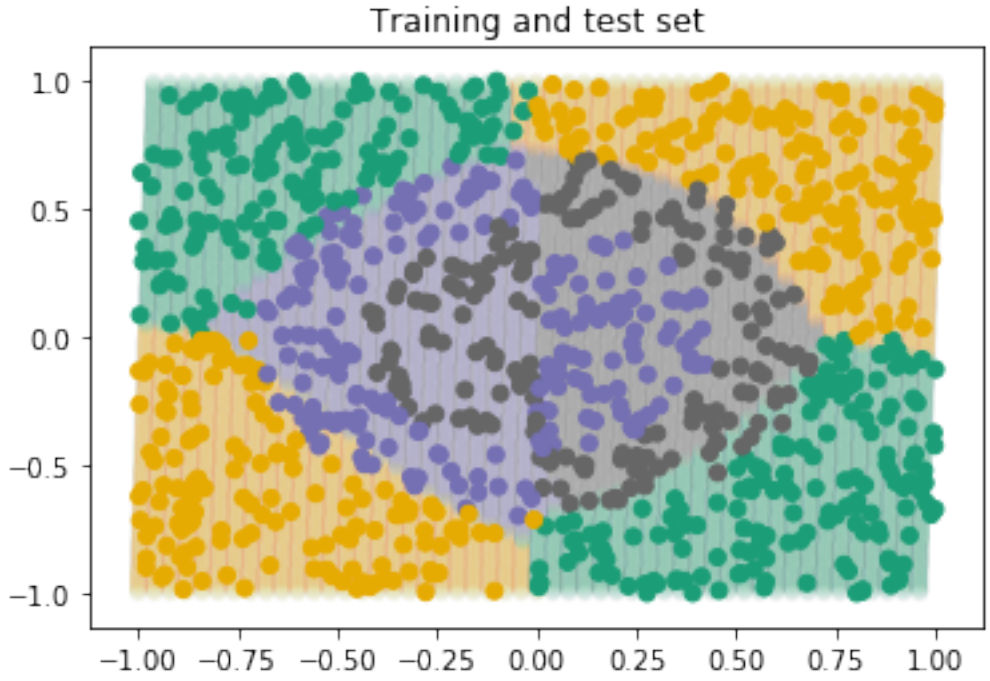

Figure 60: png

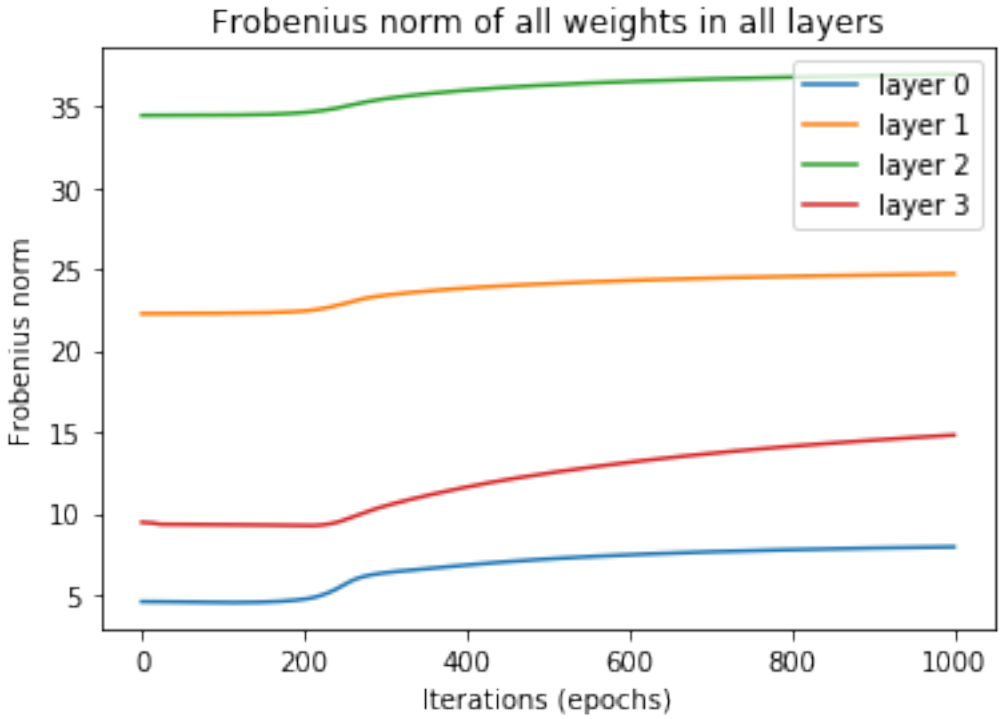

Figure 61: png

```
brain.add(Layer(256,128,'sigmoid'))
brain.add(Layer(128,4,'sigmoid'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 20/20
```

Figure 62: png

Widać, że już dla 20 epok wynik jest naprawdę dobry! Zobaczmy czy będziemy w stanie uzyskać jeszcze lepszy wynik dla większej ilości epok.

```
np.random.seed(1)
x = data_read_classification('circles',10000)
brain = Network(learning_rate = 0.1, momentum_rate = 0.8, iterations = 200)
brain.add(Layer(2,64,'sigmoid'))
brain.add(Layer(64,256,'sigmoid'))
brain.add(Layer(256,128,'sigmoid'))
brain.add(Layer(128,4,'sigmoid'))
errors = brain.train_and_evaluate(x[0],x[1],x[2],x[3])
plot_errors(errors)
plot_classification(brain, x)
weights_norms_plot(errors)
```

```
Epoch: 200/200
```

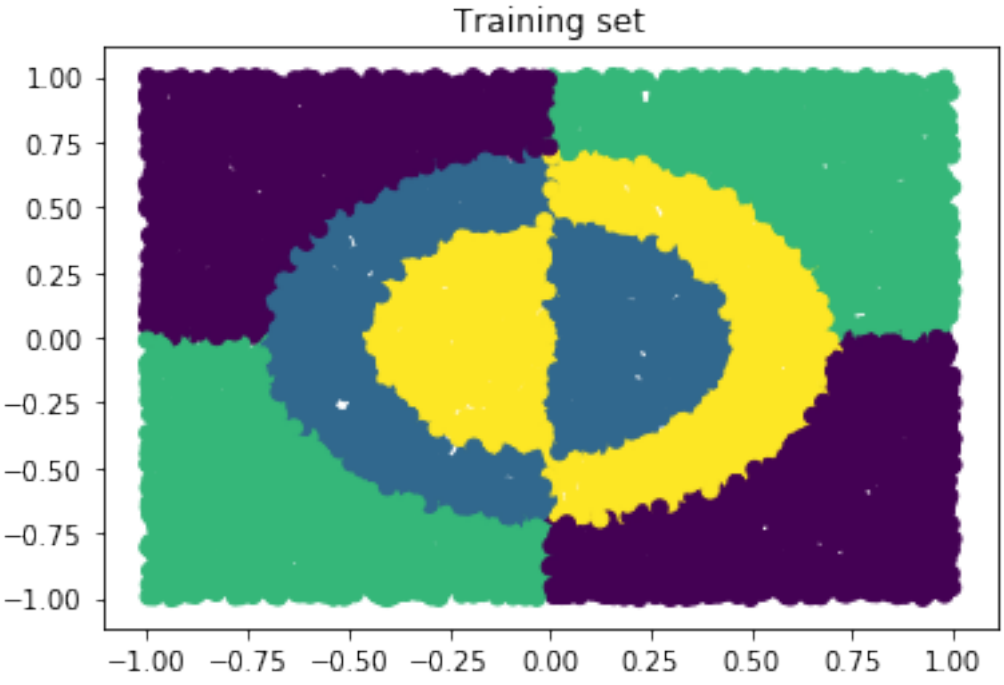

Figure 63: png

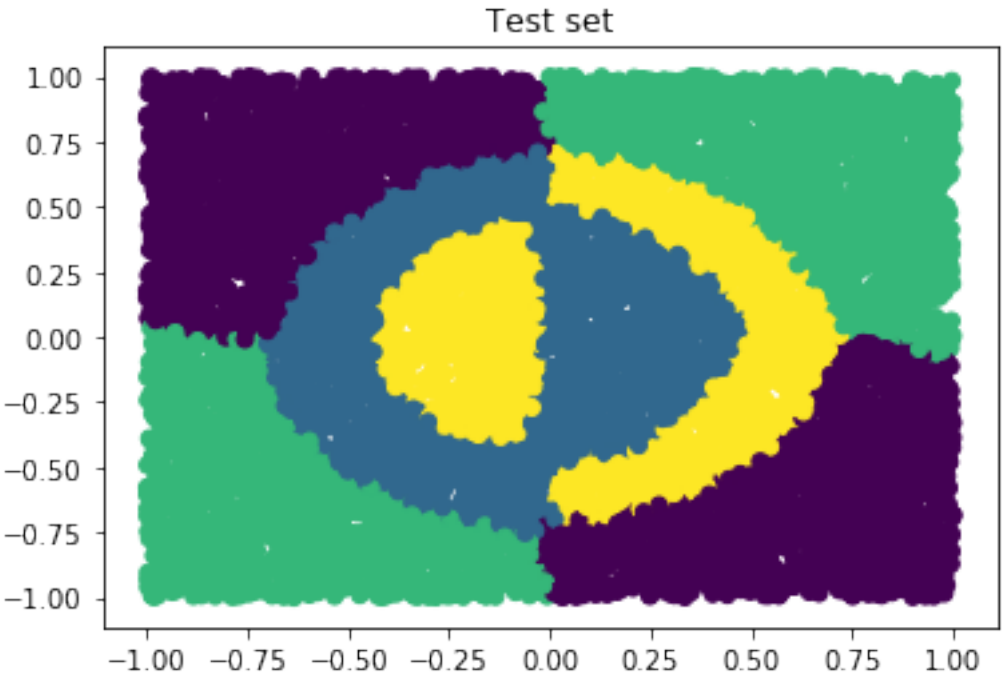

Figure 64: png

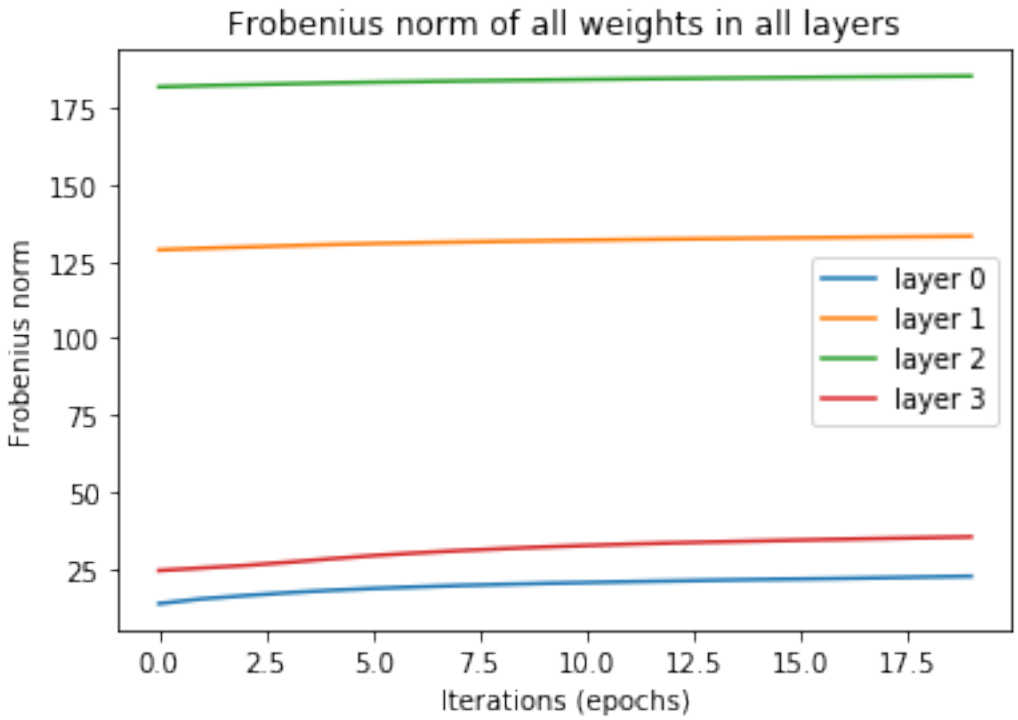

Figure 65: png

Figure 66: png

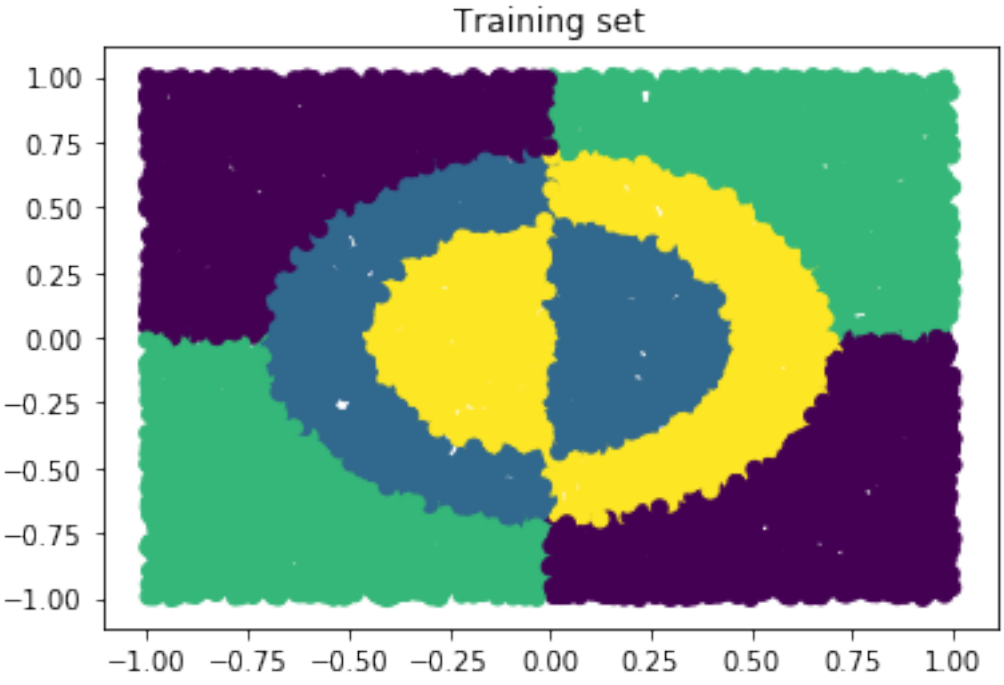

Figure 67: png

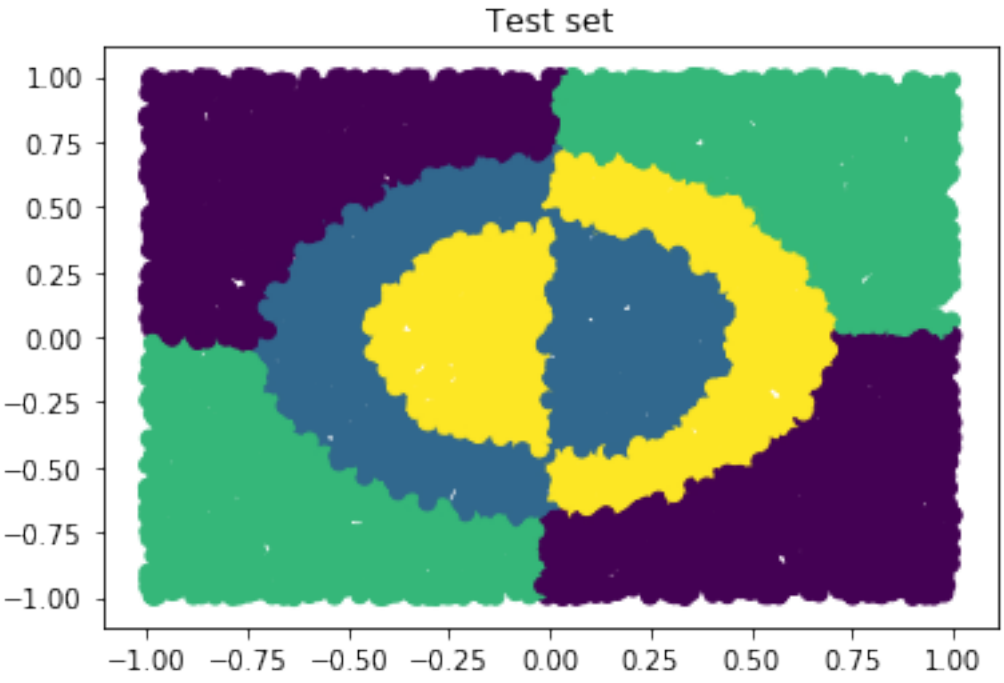

Figure 68: png

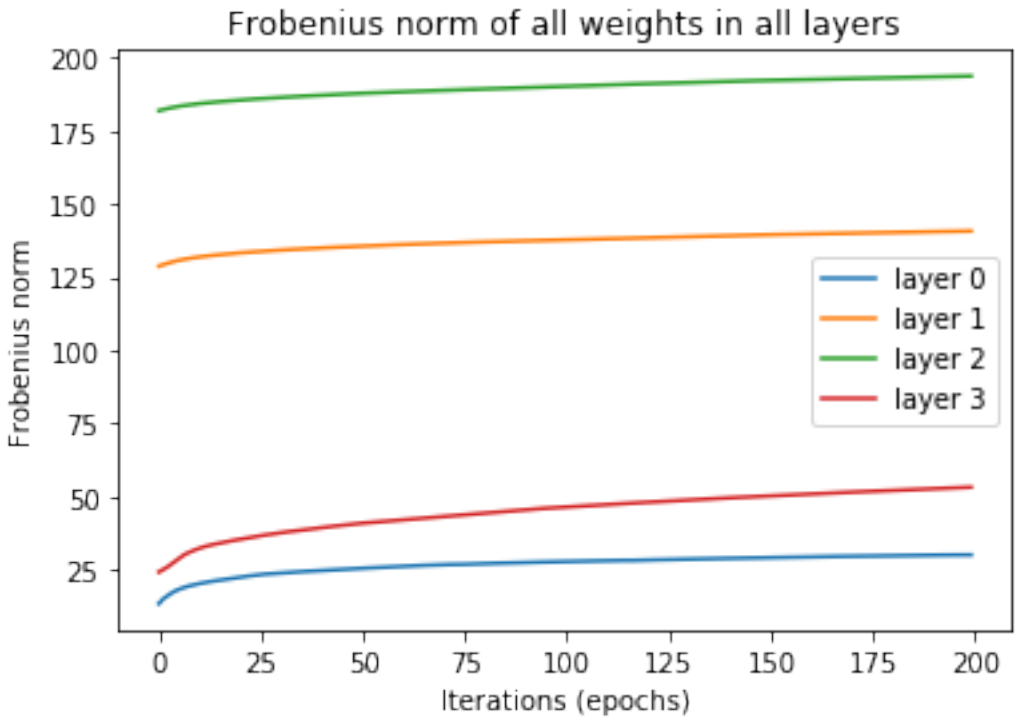

Figure 69: png

Wnioski

Po zaimplementowaniu, przeanalizowaniu i wielu eksperymentach przeprowadzonych na sieciach, możemy przyznać że jest to potężne narzędzie do rozwiązywania problemów regresji i klasyfikacji. Wystarczy tylko odpowiednio dobrać parametry do problemu.

Wpływ parametrów:

- **Learning rate** - parametr odpowiedzialny za szybkość uczenia. Zauważyliśmy, że jeżeli jest zbyt duży, funkcja błędu oscyluje wokół jakiejś wartości (minima lokalnego/globalnego) i nie można wtedy powiedzieć, że parametr został dobrze dobrany. Należy go w takim wypadku zmniejszyć. Jeżeli chodzi o zmniejszanie learning rate to równolegle, powinniśmy zwiększyć również liczbę iteracji (epok).
- **Momentum rate** - parametr odpowiadający za szybkość uczenia poprzez wykorzystanie gradientu z poprzedniej iteracji. W wielu przypadkach po prostu przyśpiesza uczenie sieci.
- **Iterations (epochs)** - parametr odpowiadający za liczbę powtórzeń procesu nauki. Najlepiej jak jest odpowiednio duży, jednak należy pamiętać aby nie doprowadzić do przeuczenia sieci. Dlatego zawsze powinniśmy rysować wykresy zbiorów treningowych/testowych jak i wykresy błędów.
- **Liczba warstw i neuronów** - im więcej warstw i neuronów tym do bardziej skomplikowanych wzorców sieć może się przystosować. Zauważyliśmy jednak, że nie zawsze im więcej tym lepiej, ponieważ dla niektórych ustawień, obszerna sieć nie pografiła poradzić sobie z bardzo prostym problemem.
- **Funkcja aktywacji** - zależnie od problemu powinniśmy odpowiednio zmieniać funkcje aktywacji poszczególnych warstw.

Ogólne uwagi:

- Warto obserwować macierze wag w poszczególnych iteracjach, aby kontrolować czy nie osiągają skrajnie dużych wartości
- Zawsze rysować sobie wyniki zbioru treningowego/testowego aby zobaczyć czy sieć nie jest przeuczona albo ma źle dobrane parametry
- Warto rysować wykres funkcji błędu i zobaczyć po której iteracji jest on praktycznie niezmienny